

Rapport du TER: SMS sécurisé sur Smart-Phones

Projet de Master 1

Auteurs :
Guillaume Jérémy
Lagny Blanche

Encadrants :
Dallot Léonard
Rojat Antoine

Résumé du document

L'objet du présent rapport est de présenter le travail effectué dans le cadre du travail d'étude et de recherche dont le sujet est "*Envoi de SMS sécurisé sur smart-phone*" par Jérémy Guillaume et Blanche Lagny. Ces travaux sont encadrés par Léonard Dallot (Chercheur en cryptologie, PRISM, UVSQ) et Antoine Rojat (Doctorant en cryptographie, PRISM, UVSQ).

Le travail demandé est de mettre en place sécurité une application permettant le chiffrement de SMS sur smart-phone et de créer un programme pour pouvoir transmettre un message sécurisé sur le système d'exploitation Android.

23 Mai 2012

Sommaire

1 Introduction

L'histoire de la téléphonie commença avec la publication d'un article de Charles Bourseul dans "l'illustration" le 26 août 1854 sous le titre "*Transmission électrique de la parole*", un article qui pose les principes de la téléphonie, mais c'est en 1863 que l'un des premiers dispositif expérimental fonctionnel fut créé, la mise en contact avec le correspondant s'effectuait alors par l'action d'une opératrice. Ensuite Almon Strowger, vers 1891, qui développa un nouveau type de téléphone, dans le but de supprimer l'intervention humaine lors d'un appel. Pour ce faire, il a créé le téléphone automatique qui établit la ligne via un numéro. Enfin la dernière évolution notable est l'arrivée des téléphones portables dans les années 80, ce qui a entraîné le développement de nombreux services annexes, notamment le service SMS. Pour finir, c'est le 23 septembre 2008 que fut lancé la première version d'Android, le seul système d'exploitation mobile permettant le développement gratuit d'application.

Le SMS est un service intégré dans les téléphones permettant d'envoyer des messages courts à un contact, via le réseau GSM. Pour cela, le protocole utilisé est "Short Message Service - Point to Point (SMS-PP)". Ce protocole utilise le principe de "Store and forward" qui pour transmettre le SMS, l'envoie à un centre SMS qui le stocke jusqu'à la réussite de la réception du message par le destinataire.

Mais nous pouvons voir que à travers ce protocole que les SMS circulent dans le réseau GSM sans aucune protection des messages lors des transmissions.

Pour pallier à ce manque de sécurité, nous proposons de chiffré les SMS grâce à la cryptographie symétrique. Pour cela nous avons créé un programme utilisant la version normalisé de RSA présent dans ANDROID et récupérant les clefs des différents utilisateurs sur un serveur de clefs que nous avons implémenté.

La suite du rapport est organisée en quatre parties. Dans la première partie, nous présenterons les programmes utilisés. Ensuite, nous présenterons toutes les notions ainsi que les choix que nous avons effectués pour le projet. Enfin nous présenterons le découpage des différentes parties du code. Pour finir, nous présenterons le programme en lui même avec un manuel d'utilisation, les améliorations que l'on peut apporter,... .

2 Programmes Utilisés

2.1 UML

UML (Unified Modeling Language) est un langage de modélisation graphique à base de pictogrammes, il est issu de la fusion des précédents langages de modélisation: Booch, OMT, OOSE. UML est maintenant un standard défini par l'Object Management Group (OMG) [Pie] dont la dernière version diffusée est UML 2.4.1 en aout 2011.
Le but de ce langage est de:

- comprendre et structurer les besoins du client;
- clarifier, filtrer et organiser les besoins;
- définir le contour du système à modéliser;
- identifier les fonctionnalités principales du système;
- permettre une meilleure compréhension du système;
- servir d'interface entre tous les acteurs du projet;

Pour cela, UML propose, depuis la version 2.3, 13 types de diagrammes dont les deux principaux (diagramme des classes et diagramme de séquence) sont définis ci-dessous.

Diagramme des classes:
Le diagramme des classes permet de modéliser les classes et interfaces d'un systèmes ainsi que les différentes relations entre les éléments le constituant. Il existe quatre types de relations qui sont:

L'héritage Elle est représenté par un trait et un triangle a une extrémité reliant deux classes. La classe fille (celle en contact avec le triangle) hérite de tous les attributs et les méthodes de l'autre classe (on dit la classe mère),

L'association Elle est représenté par un trait reliant deux classes. Elle représente une connexion sémantique entre les deux classes.

L'agrégation Elle est représenté par un trait reliant deux classes dont l'une des extrémités est un losange vide. Il représente une relation "ensemble / élément" et dont l'élément peut exister sans appartenir à un ensemble,

La composition Elle est représenté par un trait reliant deux classes dont l'une des extrémités est un losange plein. Il représente une relation "ensemble / élément" et dont l'élément ne peut exister sans appartenir à un ensemble.

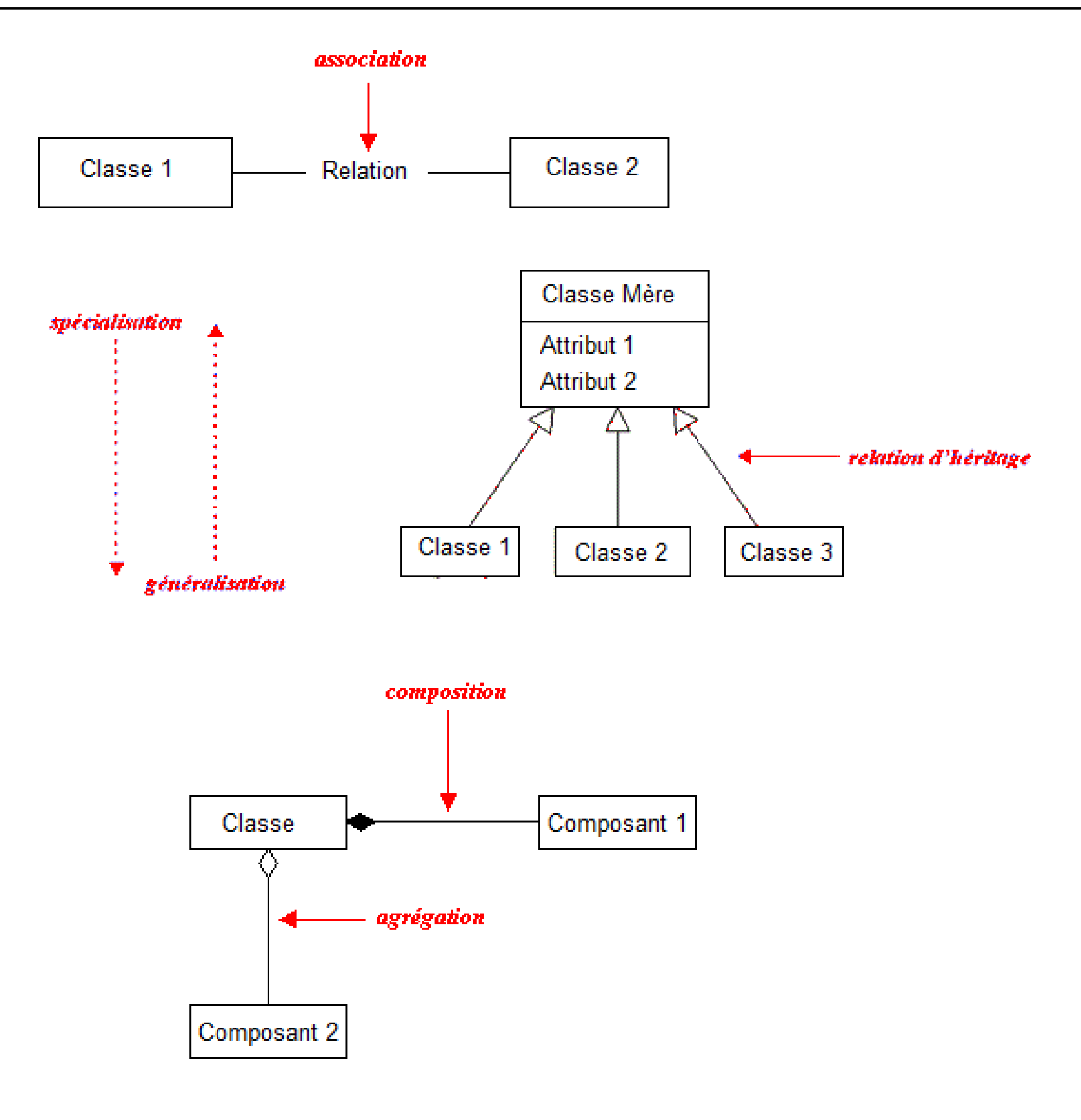


Figure 1: Présentation des relations dans un diagramme des classes

Diagramme de séquence:
Le diagramme de séquence permet de modéliser les interactions entre les acteurs et le système au cours du temps. Pour modéliser les interactions, cinq types de message ont été défini:

le message simple Un message dont l'expéditeur ne donne aucune caractéristique d'envoi ou de réception particulière.

le message avec durée de vie L'expéditeur est bloqué pendant un temps donné, mais il est libéré si au bout du temps imparti le message n'est pas pris en compte.

le message synchrone L'expéditeur est bloqué jusqu'à la prise en compte du message.

le message asynchrone Aucune interruption de l'expéditeur peut importe que le message soit traité ou non.

le message dérobant N'interrompt pas l'expéditeur et ne déclenche une opération chez le récepteur uniquement si il attendait le message.

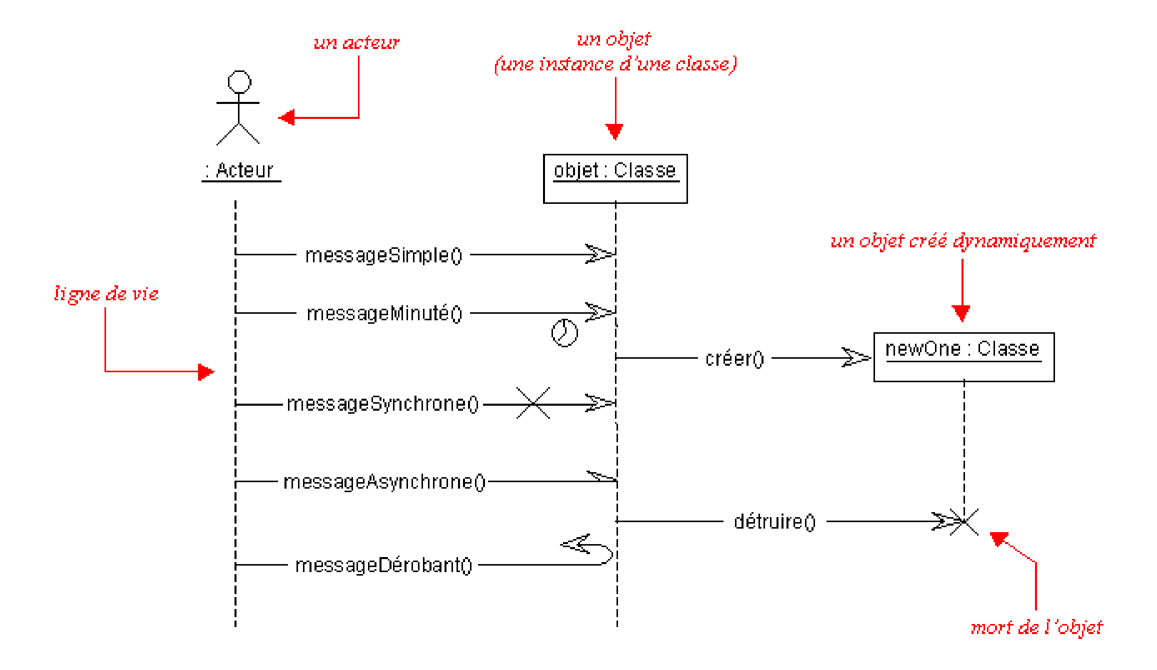


Figure 2: Présentation d'un diagramme de séquence

2.2 ANDROID

Android est un système d'exploitation (SE) open-source. Il est disponible pour des smart-phones ou des tablettes principalement, mais est aussi utilisé comme SE pour les ordinateurs, pour des MP3, des montres... Il a été annoncé le 5 Novembre 2007 par l'Open Handset Alliance, une association de plusieurs entreprises liées aux technologies mobiles (constructeurs, opérateurs mobiles, éditeurs de logiciels) dont les membres fondateurs sont Aplix, Ascender Corporation, Audience, Broadcom , China Mobile, eBay, Esmertec, Google, HTC, Intel, KDDI, Living Image, LG, Marvell, Motorola, NMS Communications, Noser, NTT DoCoMo, Inc., Nuance, Nvidia, PacketVideo, Qualcomm, Samsung, SiRF, SkyPop, SONiVOX, Sprint Nextel, Synaptics, TAT - The Astonishing Tribe, Telecom Italia, Telefónica, Texas Instruments, T-Mobile et Wind River [All07a].

Ce système d'exploitation a été créé dans le but de favoriser l'innovation sur les appareils mobiles en proposant aux développeurs un environnement ouvert où ils peuvent accéder à toutes les fonctionnalités sauf les pilotes contenus dans le noyau Linux (pilotes de l'appareil photo, de l'écran, du WiFi...) qui ne sont accessible que par le framework de développement. Il propose également aux constructeurs une plus grande liberté dans le design de leur produits. La plate-forme Android est composée d'un noyau de Linux 2.6, d'une machine virtuelle Java (Dalvik Virtual Machine), de plusieurs bibliothèques...

Elle est à ce jour en version 4.0 et la version 5.0 a été annoncée pour l'été 2012.

2.2.1 Android SDK

Le 12 Novembre 2007, une semaine après avoir annoncé l'arrivée d'Android, l'Open Handset Alliance met à disposition l'Android Software Development Kit (Android SDK) afin que les développeurs puissent avoir tous les outils nécessaires pour programmer des applications Android [All07b]. Parmi ces outils on trouve de la documentation, des exemples de codes, des APIs, des outils de débogage, un emulateur (Android Virtual Device)... La documentation est une documentation Javadoc permettant de documenter les différentes méthodes des APIs.

Dans l'Android SDK se trouve également un plugin Eclipse, Android Development Tools (ADT), qui permet de programmer des applications Android en utilisant Eclipse.

La dernière version de l'Android SDK permet aux applications créées de fonctionner sur Android 4.0, cependant, comme nous avons voulu mettre à disposition notre application pour le plus d'utilisateurs possible, elle sera compatible à partir d'Android 2.1, version qui concernait encore 5,5% des smartphones au 1er mai 2012 [Dev12]. En effet, Android assure la rétro-compatibilité des applications, c'est-à-dire qu'une application fonctionnant sur Android 2.1 fonctionnera sur Android 4.0. De plus, il n'y a pas eu de changements majeurs entre les versions 2.1 et 2.2 sur les

APIs que nous utilisons, alors qu'entre la version 1.6 et 2.1, il y en a eu, notamment sur la gestion des contacts, et nous ne voulions pas programmer en utilisant des méthodes dépréciées.

Platform	Codename	API Level	Distribution
Android 1.5	Cupcake	3	0.3%
Android 1.6	Donut	4	0.7%
Android 2.1	Eclair	7	5.5%
Android 2.2	Froyo	8	20.9%
Android 2.3 - Android 2.3.2	Gingerbread	9	0.5%
Android 2.3.3 - Android 2.3.7	Gingerbread	10	63.9%
Android 3.0	Honeycomb	11	0.1%
Android 3.1	Honeycomb	12	1.0%
Android 3.2	Honeycomb	13	2.2%
Android 4.0 - Android 4.0.2	Ice Cream Sandwich	14	0.5%
Android 4.0.3 - Android 4.0.4	Ice Cream Sandwich	15	4.4%

Tableau 1: Répartition des versions d'Android sur smartphones au 1 Mai 2012

2.2.2 La programmation Android

Une application Android est constituée de plusieurs composants : des activités, des fournisseurs de contenus, des Intents, des services... Une application n'utilisant pas forcément tous ces composants, nous allons seulement voir ceux dont nous nous sommes servis.

Les activités : Une activité correspond à une page de l'interface utilisateur. Par exemple dans notre application, la page d'accueil correspond à une activité, la page permettant d'écrire un SMS à une autre... Une activité est donc composée de deux éléments : une classe qui étend Activity (du paquet android.app) et une interface utilisateur qui peut être soit dans un fichier XML externe, soit définie dans la classe.

Les Intents : Les Intents permettent à des composants de votre application de communiquer avec d'autres composants, qu'il s'agisse d'une autre activité dans votre application ou bien d'une application différente. Ils permettent par exemple d'informer les applications d'un appel entrant ou bien que le niveau de batterie est faible... Nous allons donc utiliser les Intents lorsque nous allons changer d'activité dans notre application ou bien lorsque nous voudrons envoyer un message au système Android pour l'informer qu'une action s'est produite : par exemple, dans notre application, nous enverrons un Intent au système lorsque nous enverrons un SMS.

Les PendingIntents : Les PendingIntents sont des Intents que l'on demande à une autre application de lancer de notre part. Par exemple, les widgets sur un bureau sont exécutés par l'interface de navigation (le launcher) de la part de l'application correspondant au widget.

Les BroadcastReceiver : Les BroadcastReceiver sont des récepteurs d'Intent. Un BroadcastReceiver est constamment à l'écoute et va réagir lorsqu'il verra un Intent qui lui est destiné. En effet, un BroadcastReceiver possède un filtre d'Intent. Dans notre application par exemple, nous avons créé un BroadcastReceiver qui réagit lorsqu'il voit un Intent dont l'action est une chaîne valant "com.uvsq.tersms.envoyer" (qui correspond à l'envoi d'un SMS à partir de notre application).

Le Manifest : Le Manifest Android n'est pas exactement un composant de votre application. Il s'agit en réalité d'un fichier contenant la description de votre application : sa version, les activités, les BroadcastReceiver, les permissions dont elle a besoin...

SQLite Dans une application, vous pouvez créer votre base de données SQLite pour stocker des informations. Dans notre programme, par exemple, nous en créerons une pour stocker les certificats de nos contacts. SQLite est une base de données relationnelles légère, gratuite et open-source. Un des intérêts de SQLite par rapport à d'autres bases de données est qu'elle ne nécessite pas de serveur pour s'exécuter : tout se fait dans l'application.

2.2.3 ECLIPSE

Eclipse est un Environnement de Développement Intégré (ou IDE pour Integrated Development Environment), c'est-à-dire qu'il contient un éditeur de code, un compilateur et un débogueur. Il est de plus open-source sous licence EPL (Eclipse Public License). Il a été conçu par IBM en Novembre 2001, et est géré depuis 2004 par l'Eclipse Foundation, consortium d'éditeurs de logiciels [Fou04]. Eclipse est multi-plateformes : il fonctionne sur Linux comme sur Windows ou Mac OS. Il a été à l'origine créé pour le développement dans le langage Java, mais est aujourd'hui multi-langages, grâce à un système de plugins.
Le plugin Android pour Eclipse nous a également permis d'avoir directement accès à un émulateur pour pouvoir tester notre application sur différentes versions d'Android, afin de vérifier sa compatibilité.

3 Cryptographie et Architecture de confiance

Pour permettre de chiffrer des SMS, nous avons dû étudier les deux types de cryptographie que sont la cryptographie symétrique et l'asymétrique [Buc06],[eCW11]. Mais aussi des notions supplémentaires pour palier au inconvénient de la cryptographie choisi tel que les signatures ou les infrastructures de confiance pour la cryptographie asymétrique. Pour ce faire, nous allons d'abord présenter les deux types de chiffrement, expliquer le choix puis .

3.1 Cryptographie symétrique

La cryptographie symétrique est la plus ancienne forme de chiffrement, des traces de son utilisation ont été découverte sur des documents de l'époque de Jules César. Le principe fondamentale de ce type de cryptographie est que l'on utilise une unique clef, que l'on partage (entre deux personnes), pour chiffrer et déchiffrer un message, c'est la connaissance de cette clef qui permet d'assurer l'identité de l'émetteur. Ainsi si un pirate découvre la clef, il pourra écrire un message en usurpant l'identité d'une des deux personne.

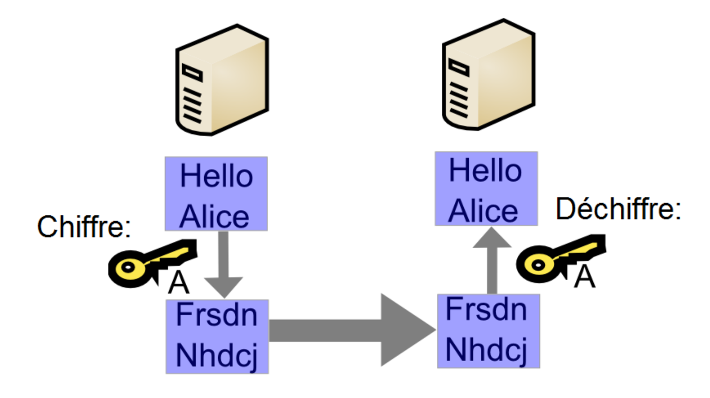


Figure 3: Principe de la cryptographie symétrique

Avantages:
En général, les algorithmes de chiffrement symétrique sont souvent très bien adaptés à l'architecture du processeur utilisé, ils sont donc très bien appropriés au chiffrement de grande quantité de données très rapidement. De ce fait, pour un niveau de sécurité donné, la cryptographie symétrique est beaucoup plus rapide.
D'autre part, ils utilisent des clefs de tailles inférieures, pour un niveau de sécurité donné, par exemple l'AES utilise des clefs de taille maximales de 256 bits tandis que les clefs pour RSA sont de 1024 bits.

Inconvénient:
Le problème de la cryptographie symétrique est le nombre de clef à gérer lorsque l'on souhaite communiquer de façon confidentielle deux à deux (communiquer avec une personne précise sans que les autres personnes du système ne puissent le déchiffrer) dans un réseaux contenant n personnes. Dans ce cas, il nous est nécessaire de posséder $n * (n - 1)/2$ clefs.

3.2 Cryptographie asymétrique

C'est en 1976 que le concept de cryptographie le concept de cryptographie asymétrique a émergé suite aux travaux de Diffie et de Hellman qui proposèrent un protocole permettant de s'entendre sur une clef sans avoir besoin d'un secret commun au préalable. Cependant il a fallu attendre 1978 pour que Rivest, Shamir et Adleman publient le schéma RSA, permettant d'échanger un message chiffré sans avoir besoin d'un secret commun. Dans la cryptographie asymétrique, nous avons la présence de deux clés liées, l'une est publique et permet de chiffrer un message et la deuxième est privée et permet de déchiffrer ce message. Ainsi pour que Bob transmette un message à Alice, Il sera nécessaire qu'Alice possède deux clés, l'une transmise à Bob via le canal public et la deuxième qu'elle conserve de manière privée.

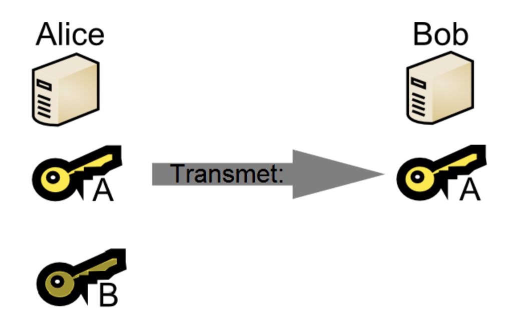


Figure 4: Répartition des clefs dans la cryptographie Asymétrique

Pour transmettre le message à Alice, Bob va chiffrer le message avec la clé publique d'Alice et envoyer le message sur le canal public, quand Alice reçoit le message, elle le déchiffre avec sa clé privée (elle est la seule à la connaître donc la seule à pouvoir déchiffrer le message).

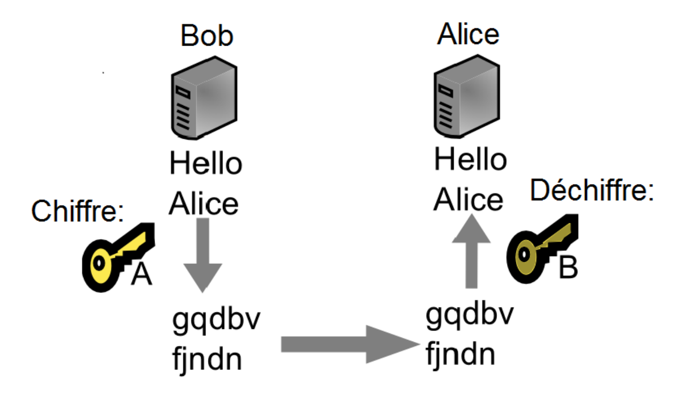


Figure 5: Représentation du fonctionnement de la cryptographie symétrique

Avantages:
L'un des avantages de la cryptographie asymétrique est la transmission des clefs, contrairement à la cryptographie symétrique, il n'est pas nécessaire de posséder un canal privé pour transmettre la clef publique, puisque un pirate en obtenant la clef transmise, ne pourra qu'écrire un message. Le deuxième avantage est le nombre de clef, ainsi pour pouvoir communiquer de façon confidentielle deux à deux, nous ne sommes plus obligé de posséder $n * (n-1)/2$ clefs mais $n + 1$ clefs suffisent (notre clef publique, notre clef privé et la clef publique de chaque personne présente dans le système). Pour finir, l'arrivée de nouveau utilisateurs ne demande que peu d'effort, il suffit de créer un couple de clef et de diffuser la clef publique créée.

Inconvénients:
Il y a deux problèmes avec la cryptographie asymétrique, le premier est que toutes les personnes du système peuvent nous envoyer des messages chiffrés à partir d'une clef or il est nécessaire de connaitre l'identité de l'émetteur pour pouvoir juger la pertinence du message ce qui n'est possible qu'en utilisant la notion de signature. Le deuxième problème est que lorsque l'on récupère une clef publique, nous ne pouvons savoir si la clef provient effectivement d'Alice ou si elle provient d'un pirate ce faisant passer pour Alice. Pour résoudre ce problème, la solution la plus courante est l'utilisation des modèles de confiance qui permettent de vérifier l'identité de l'émetteur d'une clef.

3.3 Choix du système de chiffrement

Dans ce projet, nous allons avoir des utilisateurs qui s'ajoute régulièrement et voulant communiquer avec une multitude de personne de manière confidentielle deux à deux de ce fait, nous avons choisi d'utiliser la cryptographie asymétrique. De plus la vitesse de la cryptographie symétrique n'est pas un réel avantage dans ce projet puisque nous n'allons chiffrer que peu de long message.

3.4 L'algorithme RSA:

L'algorithme RSA a été publié en 1978 par Rivest, Shamir et Adleman dans leur article A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Ce système cryptographique asymétrique est aujourd'hui le plus utilisé dans l'industrie. Sa popularité vient à la fois du fait qu'il a été un des premiers schémas à proposer une instanciation concrète de la cryptographie asymétrique et de la simplicité de son implémentation. Ce schéma fonctionne dans un anneau du type $\mathbb{Z}/n\mathbb{Z}$; autrement dit les messages clairs et les messages chiffrés sont des éléments d'un tel anneau.

Création des clés:
Les clefs de RSA sont formées de

- deux nombres premiers distincts p et q,
- le produit $N = p * q$,
- les entiers distincts e et d premier avec $(p - 1)$ et $(q - 1)$ tel que $e * d = 1mod(p - 1)(q - 1)$.

et sont générés selon l'algorithme suivant:

Algorithme 1 Génération des Clefs avec RSA

$p \leftarrow nbr_premier()$ //Stocke un nombre premier
$q \leftarrow nbr_premier()$ //Stocke un nombre premier
$N \leftarrow p * q$
$e \leftarrow Alea_premier((p - 1), (q - 1))$
$d \leftarrow Euclide_Etendu((p - 1), (q - 1), e)$ //Algorithme qui calcule d tel que e*d = 1 mod (p-1)(q-1)
Return (N, e) la clef publique et (N, d) la clef privé.

Chiffrement:
Le chiffrement de RSA se fait en passant le message à la puissance e modulo N

Algorithme 2 Chiffrement des messages avec RSA

Input: La clef publique (N, e), le message M
$C \leftarrow M^e \mod (N)$
Return Le chiffré C

Déchiffrement:
Le déchiffrement de RSA se fait en passant le chiffré à la puissance d modulo N

Algorithme 3 Déchiffrement des messages avec RSA

Input: La clef privée (N, e), le message C
$M \leftarrow C^d \mod (N)$
Return Le message M

Avantages:
L'un des avantages du crypto-système RSA est d'avoir un développement simple, il est composé d'algorithme facile à implémenter, de plus il est le système de cryptographie asymétrique le plus utilisé dans le monde et il est considéré comme un standard en Cryptographie. Enfin il déjà est implémenté dans divers langages tel que le Java, php (openSSL) mais aussi Android.

Inconvénient:
L'inconvénient principal est le fait que la sécurité du système n'est que conjecturé aussi sûr que le problème de la factorisation.

Conclusion:
Pour ce projet, nous avons décidé d'utiliser le crypto-système RSA parce qu'il est suffisamment rapide pour l'usage effectué dans le projet et aussi parce qu'il est déjà implémenté sur ANDROID.

RSA dans notre application Pour la programmation du système RSA, nous avons utilisé des classes qui étaient déjà définies dans les paquets java.security (pour la génération des clefs) et javax.crypto (pour le chiffrement et déchiffrement). Tout le code correspondant à cette partie se trouve dans la classe `RSA` de notre code.
La première méthode de cette classe a pour signature `static KeyPair generateKeyPair()`. Elle est utilisée pour générer une paire de clefs.

```
KeyPairGenerator kpg=KeyPairGenerator.getInstance("RSA");
//on prend des clefs de taille 1024 bits et un modulo = 65537
RSAKeyGenParameterSpec rsa=new RSAKeyGenParameterSpec(1024,RSAKeyGenParameterSpec.F4);
kpg.initialize(rsa);
kp=kpg.generateKeyPair();
```

Nous avons utilisé l'algorithme de génération des clefs déjà implémenté dans le paquet java.security en faisant appel aux classes `KeyPairGenerator` et `RSAKeyGenParameterSpec`. La première de ces classes permet de générer une paire de clef publique et privée mais n'est pas spécifique à RSA, c'est pourquoi nous avons eu besoin de la deuxième afin de préciser les spécificités que nous désirions. Celles-ci sont passées en argument au constructeur de `RSAKeyGenParameterSpec` : le premier est la taille des clefs (en bits), le second est la valeur de l'exposant public, pour lequel nous avons utilisé une valeur prédéfinie qui est `RSAKeyGenParameterSpec.F4` et qui correspond à 65537, soit $2^{16} + 1$.

La seconde méthode est `public static String RSAEncode(String mess, PublicKey public1)`. Elle est utilisée pour chiffrer le message `mess` avec la clef publique `public1`.

```
byte[] messByte, smsByte, encodeByte = null;
String chiffre= null;
Cipher c=Cipher.getInstance("RSA/ECB/PKCS1Padding");
c.init(Cipher.ENCRYPT\_MODE, public1);
//on d\'ecoupe en bloc de 117 bytes
for(int i=0; i<mess.length(); i+=116){
        if((i+116)>=mess.length()){
                messByte= mess.substring(i).getBytes("ISO-8859-1");
        }
        else{
                messByte= mess.substring(i, i+116).getBytes("ISO-8859-1");
        }
        encodeByte= c.doFinal(messByte);
        if(i==0){
                //on rajoute un octet au d\'ebut pour diff\'erencier les messages chiffres
                smsByte=new byte[encodeByte.length+1];
                smsByte[0]=127;
                for(int j=0; j<encodeByte.length; j++){
                        smsByte[j+1]=encodeByte[j];
                }
                chiffre=new String(smsByte, "ISO-8859-1");
        }
        else{
                chiffre+=new String(encodeByte, "ISO-8859-1");
        }
}
```

Pour cette méthode, nous utiliserons la classe `Cipher` du paquet javax.crypto. Cette classe va nous permettre de chiffrer le message à envoyer. Pour cela, nous initialisons d'abord un `Cipher`: le constructeur doit prendre en paramètre une chaîne représentant soit le nom de l'algorithme à utiliser (RSA, AES,...) soit l'algorithme, le mode et le padding. C'est cette dernière méthode que nous avons utilisé : nous instancions donc un nouvel objet `Cipher` qui utilisera l'algorithme RSA, le mode ECB et le padding PKCS#1 (v1.5).
Nous devons ensuite découper le message en bloc de plus petite taille. Pour un RSA classique avec une taille de clefs de 1024 bits, la taille maximum d'un bloc est de 128 octets. Or ici nous utilisons en plus un padding qui rajoute 11 octets au bloc de données, nous avons donc une taille maximum de bloc de 117 octets. C'est pourquoi nous découpons le message en bloc de 117 octets et chiffrons ces blocs un par un.
Le chiffrement effectif des données se fait grâce à la méthode `doFinal` de la classe `Cipher`. Cette méthode prenant en argument un tableau d'octet, nous devons convertir le message: c'est le rôle de la méthode `getBytes` de la classe `String`. Nous avons précisé un encodage en argument car l'encodage par défaut est l'UTF-8 et nous avons découvert que la présence d'octets représentant des caractères non affichable (DEL, NUL...) faisait échouer le déchiffrement dans cet encodage.

Une fois le chiffrement fait, nous rajoutons un octet au début du message afin de différencier les messages chiffrés des clairs lors de l'affichage (et ainsi éviter le déchiffrement d'un message clair). Cet octet doit correspondre à un caractère qui ne peut pas être présent dans un message clair, nous avons donc choisi 127, qui correspond à DEL.

Enfin, la dernière méthode de cette classe est `public static String RsaDecode(String chiffre, PrivateKey private1)` qui servira à déchiffrer des messages.

```
byte[] smsByte, messByte =null;
String dechiffre="";
Cipher c;
c = Cipher.getInstance("RSA/ECB/PKCS1Padding");
c.init(Cipher.DECRYPT\_MODE, private1);
chiffre=chiffre.substring(1);
for(int i=0; i<chiffre.length(); i+=128){
        if(i+128>=chiffre.length()){
                smsByte= chiffre.substring(i).getBytes("ISO-8859-1");
        }
        else{
                smsByte= chiffre.substring(i, i+128).getBytes("ISO-8859-1");
        }
        messByte=c.doFinal(smsByte);
        dechiffre+=new String(messByte, "ISO-8859-1");
```

Comme pour la méthode précédente, nous utilisons la classe `Cipher` avec le même triplet algorithme, mode et padding. Nous enlevons ensuite le premier octet du message qui est l'octet 127 ajouté pour la différenciation des messages clairs et chiffrés; puis nous découpons le message en bloc de 128 octets pour les déchiffrer un par un. En effet, le chiffrement d'un bloc en RSA avec une taille de clefs de 1024 bits sera un bloc de 128 octets qui sera éventuellement concaténé avec un autre bloc si le message est trop long, c'est pourquoi nous devons les séparer avant de les déchiffrer.

3.5 Les signatures

La signature est un mécanisme permettant d'assurer l'intégrité du message et d'authentifier l'auteur de ce message.

Schéma de signature:
Pour créer une signature, l'émetteur va utiliser le message sur lequel il va appliquer la primitive de la fonction de chiffrement (fonction qui "annule" le chiffrement d'un message avec la clef publique) qui utilise comme argument le message et la clef privé de l'émetteur. Ensuite, l'émetteur va transmettre le message chiffré et la signature. Enfin après la réception, le réceptionnaire va chiffrer la signature avec la clef publique de l'émetteur, si l'on obtient le message, alors l'on est sûr de l'identité de l'émetteur (dans l'hypothèse où l'on a certifié la possession de la clef publique), car seul la personne connaissant la clef privé associée à la clef publique de chiffrement utilisé à pu créer cette signature, de plus pour modifier le message aléatoire, il faut aussi changer la signature ce qui n'ai pas faisable pour un pirate sans changé l'identité de l'émetteur.

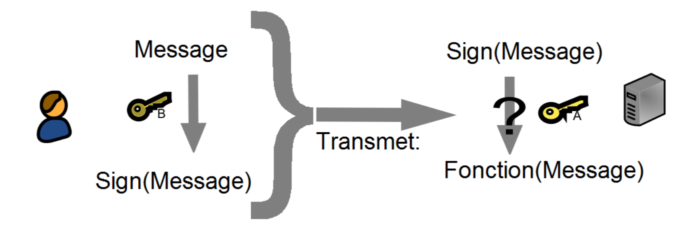


Figure 6: Principe de fonctionnement de la signature

Mise en place:
Dans notre cas, nous utilisons le crypto-système RSA dont la primitive de la fonction de chiffrement est la fonction de déchiffrement ce qui permet d'éviter un rajout de code. De plus, la signature est créé a partir du message chiffré transmit en utilisant la fonction de hachage SHA1.

La signature dans notre application

La signature d'un message est implémentée dans la classe `Signer` de notre code. Cette classe utilisera la classe `Signature` du paquet java.security dans laquelle sont implémentés plusieurs algorithmes de création et de vérification de signature, dont celui utilisant la fonction de hachage SHA-1 et l'algorithme RSA que nous utiliserons.
Cette classe contient donc 2 fonctions: `public static String signeMess(String mess, PrivateKey privKey)` et `public static boolean verifieSignature(String sign, String mess, PublicKey pubkey)`.

La fonction `signeMess` prend en argument le message à signer et la clef privée de l'utilisateur qui va lui servir de clef de signature.

```
Signature s = Signature.getInstance("SHA1withRSA");
s.initSign(privKey);
s.update(mess.getBytes("ISO-8859-1"));
byte[] signature = s.sign();
String signatureString=new String(signature, "ISO-8859-1");
sign=SIGNATURE+signatureString;
```

Nous instancions tout d'abord un objet `Signature` en lui indiquant que l'algorithme à utiliser est SHA1withRSA.
Nous initialisons ensuite la clef qui servira à chiffrer le haché avec la clef privée de l'utilisateur passée en argument.
Nous passons ensuite le message à signer qui sera un tableau d'octet. Comme pour notre classe `RSA`, nous utilisons l'encodage ISO-8859-1 afin d'éviter toute erreur lors de la vérification de la signature.
Enfin, nous signons le message grâce à la méthode `sign()` de la classe `Signature` et convertissons le résultat en String afin qu'il soit présentable dans le message. De plus, nous rajoutons au début de la signature une chaîne que nous avons défini comme étant égale à "–SIGN–". Cette chaîne nous permettra de savoir si un message est signé ou non et de pouvoir distinguer la signature du reste du message.

La fonction `verifieSignature` prend en argument la signature à vérifier (sign), le message (mess) et la clef publique de la personne qui a signé le message (pubkey).

```
sign= sign.substring(SIGNATURE.length());
s = Signature.getInstance("SHA1withRSA");
s.initVerify(pubkey);
s.update(mess.getBytes("ISO-8859-1"));
return s.verify(sign.getBytes("ISO-8859-1"));
```

Tout d'abord, il faut supprimer la chaîne SIGNATURE, qui nous sert de délimiteur, de la signature. Ensuite, nous utilisons la classe `Signature` et nous instancions notre objet avec le même algorithme que précédemment : SHA1withRSA. En revanche, nous n'initialisons pas la clef grâce à la fonction `initSign` mais grâce à `initVerify` qui initialise la clef dans le but de vérifier une signature. Nous initialisons ensuite le contenu du message qui a été signé et nous appelons la fonction `verify` qui prend en argument la signature du message et qui renvoie vrai si la signature est vérifiée et faux sinon.

3.6 Gestion des clefs

Nous devons maintenant trouver un moyen pour qu'un utilisateur puisse récupérer la clef publique de la personne à qui il veut envoyer un message. Nous pouvons mettre cette clef à disposition sur un serveur de clefs, mais il faut alors un moyen de s'assurer qu'il s'agit bien de la clef publique de l'utilisateur voulu: en effet, si Charlie met sa propre clef publique sur le serveur en se faisant passer pour Bob, alors il pourra lire les messages adressés à Bob.
Nous avons étudié les trois modèles classiques de ces modèles avant de les comparer entre eux, pour enfin choisir celui qui nous paraissait le plus adapté à notre projet.

3.6.1 Chain Of Trust

Le principe du modèle Chain Of Trust est le suivant: supposons qu'une nouvelle utilisatrice, Alice, veuille prouver la possession de sa clef publique. Elle doit alors envoyer un document contenant des données permettant de l'identifier (nom, mail, téléphone,...) ainsi que sa clef publique si c'est elle qui l'a générée. Une Autorité de Certification (AC), va alors vérifier ces informations et, si elle est convaincue qu'il s'agit bien d'Alice, va générer un certificat qu'elle signera. Elle va ensuite récupérer le certificat signé par l'AC [Ben].
Supposons maintenant que Bob, un autre utilisateur, veuille envoyer un message à Alice. Il récupère donc son certificat et regarde le nom de l'Autorité de Certification qui l'a signé afin de pouvoir récupérer la clef publique de l'AC. Une fois cette clef récupérée, il vérifie la signature du certificat afin de vérifier son intégrité. Ensuite, deux

cas se présentent: soit Bob connaît l'AC et lui fait confiance, soit il ne la connaît pas. Dans le premier cas, il peut alors envoyer un message chiffré à Alice avec sa clef publique qu'il a récupéré. Dans le deuxième cas, il doit regarder qui a signé le certificat de l'AC. S'il s'agit d'un certificat signé par une AC qu'il connaît, alors il peut s'arrêter et envoyer le message à Alice, sinon, il continue à remonter la "chaîne" certificats jusqu'à trouver une AC qu'il connaît ou bien jusqu'à tomber sur une autorité racine, c'est à dire une autorité qui a elle même signé son certificat et dans laquelle nous devons avoir confiance (sinon la sécurité de la chaine est détruite).

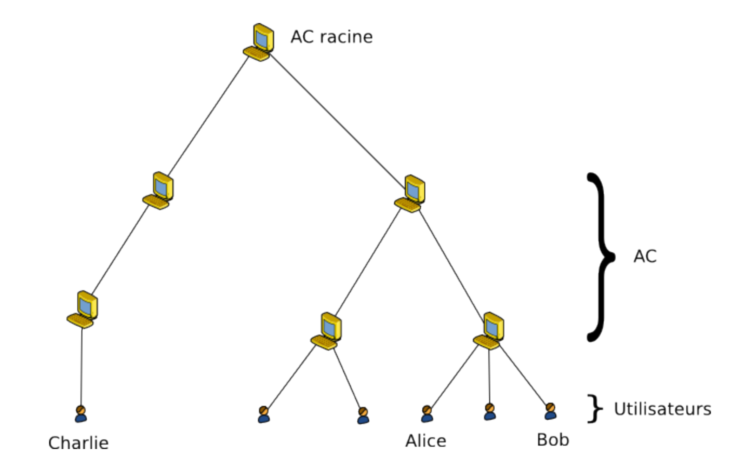


Figure 7: Ici, Alice fait confiance à Bob, car son certificat a été signé par une AC en qui elle a confiance. Elle fait également confiance à Charlie car elle peut trouver une AC qui fait à la fois confiance à son AC et à celle de Charlie

Fonctionnalités:

Récupération des clefs: Bob effectue une demande à Alice pour récupérer son certificat. Alice va lui transmettre puis à la réception de celui-ci, il va vérifier la signature.

Révocation: Pour révoquer son certificat, Alice va envoyer un message à l'AC. La révocation s'effectuera en mettant sur la liste de révocation le certificat. Ainsi lors de l'utilisation de la clef, on vérifiera si le certificat n'est pas dans la liste de révocation et dans le cas contraire, une demande de récupération de la clef sera effectué.

Génération des certificats: Pour générer son certificat, Alice va transmettre à l'AC des informations sur son identité ainsi que sa clef publique. L'AC en recevant la demande va vérifier l'identité d'Alice, si la vérification est positive, elle va compléter le certificat et le signer pour le retransmettre à Alice.

La limite:
La limite principale du chain of trust est de devoir faire confiance à toutes les AC de la chaine or si une autorité ne se protège pas correctement, alors tout les "descendants" dans la chaine sont compromis alors qu'il n'y a aucun moyen de connaitre la sécurité de l'AC.

3.6.2 Web Of Trust

Principe: Le modèle Web Of Trust est assez différent du premier modèle: il s'agit d'un modèle de confiance décentralisé, c'est-à-dire qu'un utilisateur peut signer la clef d'un autre, il n'y a plus d'Autorité de Certification [AAR]. Au fur et à mesure, des signatures s'ajouteront, soit grâce à des key signing party (où des gens s'échangent en main propre des hachés de leur clef publique, ce qui permet de vérifier leur certificat et donc de le signer), soit parce qu'il s'agit de quelqu'un que vous connaissez déjà. Ces signatures permettent aux personnes cherchant votre clef publique de vérifier l'authenticité de votre clef. Il existe plusieurs niveaux de confiance qui sont: on a confiance dans sa clef et dans les clefs qu'il nous transmet, on a confiance dans sa clef mais pas dans les clefs qu'il nous transmet et enfin on ne fait pas du tout confiance. Supposons maintenant qu'un utilisateur, Bob, veuille envoyer un message à Alice: il doit alors chercher sa clef. Bob va alors regarder les signatures de ce certificat et chercher quelqu'un qu'il connaît et en qui il a confiance parmi les signataires. S'il n'en existe aucun, il va alors chercher parmi les signataires des clefs des premiers signataires... jusqu'à ce qu'il trouve quelqu'un en qui il a confiance. Chaque utilisateur est libre de déterminer des paramètres tels que le nombre de liens maximums dans un chemin le reliant à un autre utilisateur, ou le nombre minimum de chemins différents le menant à un utilisateur,...

Fonctionnalités:

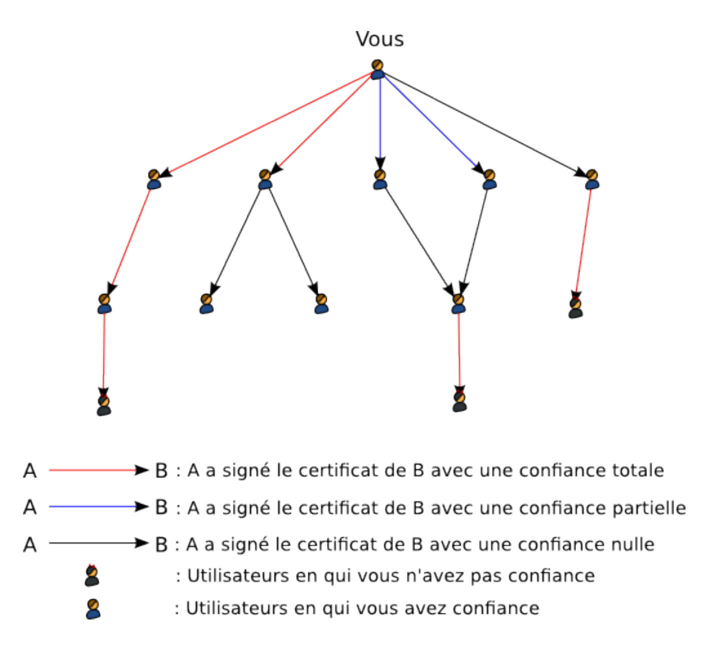


Figure 8: Organisation d'un réseau Web of Trust du point de vue d'un utilisateur

Récupération des clefs: elle s'effectue en demandant au possesseur de lui transmettre ainsi que les signatures associées de là il va créer le "chemins des personnes" et préciser le niveau de confiance de la clef (en fonction du nombre de chemins, du nombre de personnes qui constituent le chemin, du niveau de confiance donné à chacune...).

Révocation des clefs: La révocation d'une clef s'effectue en ne la donnant plus lors d'une demande. Cette suppression entraîne la perte, pour toutes les clefs signés par la personne dont les clefs ont été révoquées, de cette signature.

Génération des signatures: La génération d'une signature est faite après la vérification de l'identité physique du possesseur. Chaque personne faisant confiance à cet utilisateur peut ensuite rajouter sa signature et indiquer le niveau de confiance.

La limite:
Le gros problème du Web of Trust est la révocation des clefs, en effet, il est commun dans un tel schéma de trouver des clefs signés grâce à une clef révoqué. Ce qui a pour conséquence de fausser la confiance en cette clef.

3.6.3 CAcert

Principe: CAcert est un système hybride entre le Web of Trust et le Chain of Trust. En effet, la structure globale est identique à celle d'un modèle Chain of Trust, mais il y a la présence de "super utilisateurs" (Assurer) qui peuvent signer les certificats des utilisateurs après les avoir rencontrer et ainsi leur faire gagner des points d'assurance. Au bout d'un certain nombres de ces points, votre certificat pourra avoir une durée de validité supérieure, et avec encore plus de point, vous pouvez passer le test vous permettant de devenir vous même un super-utilisateur. Il faudra repasser ce test régulièrement si vous voulez garder votre statut. De plus, dans ce système, l'autorité de certification va automatiquement signer un certificat à partir du moment où l'adresse email est valide [CAc].

Fonctionnalités:

Récupération des clefs: elle s'effectue via la récupération d'un certificat signé au prés du possesseur, puis on analysera la signature.

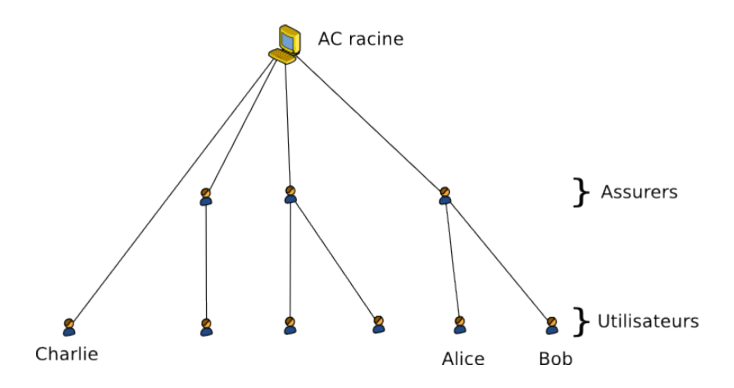


Figure 9: Organisation d'un réseau CAcert

Révocation des clefs: deux cas se présentent: soit l'utilisateur n'est pas un assureur, donc on peut révoquer son certificat sans conséquence, soit il s'agit d'un Assureur, et dans ce cas, il faut retirer tous les points d'assurance qu'il a donné aux autres utilisateurs. Si cela entraîne la perte du statut d'assureur pour un utilisateur, il faut enlever les points que celui-ci à donner à d'autres utilisateurs et ainsi de suite.

Génération des certificats: La génération du certificat est faite lors de la récupération de la clef publique de l'utilisateur sur le serveur. Un champs "assurance point" dans le certificat permet de donner des points à l'utilisateur (seulement pour les Assureur).

3.6.4 Avantages et inconvénients

Chain of trust:

Avantage: Il est très flexible, et autorise une automatisation du système de signature.

Inconvénient: Il faut faire confiance à l'autorité de certification et aux utilisateurs.

Web of trust:

Avantages: Système décentralisé et un accès à un nombre de clefs important rapidement, ne nécessite pas d'autorité.

Inconvénient: Révocation des clefs compliquées à mettre en place (recherche des différents chemins de confiance,...).

CAcert:

Avantage: Les mêmes que pour le Chain Of Trust et permet en plus de déléguer la vérification des identités.

Inconvénient: Il faut vérifier que les Assurer sont fiables, révocation des clefs compliqués.

3.6.5 Choix final

Dans ce projet, on peut se permettre de faire un système laxiste puisque l'on va vérifier le numéro de téléphone au niveau de l'application. De plus, la révocation des clefs doit être simple car il n'est pas rare de se faire voler son téléphone. Enfin, nous devons pouvoir mettre en place un système simple pour pouvoir effectué de fréquente misa à jours des clefs. De ce fait, nous avons décidé d'utiliser le modèle de Chain of trust en utilisant un unique serveur (qui va servir d'autorité racine, de serveur de stockage et de liste de révocation).

3.7 Envoi du message

Envoi des SMS sous Smart-phone:
L'envoi des SMS sous smart-phone s'effectue selon le protocole "Short Message Service - Point to Point" défini dans la norme mobile GSM 03.40 permettant l'envoi de message court à travers les canaux de signalisations [eFeT09]. Ce protocole s'effectue grâce à deux services:

- le premier est le service Short Message Mobile Terminated Point-to-Point (SMMT) qui permet la transmission du message vers un serveur et la gestion de l'acquittement de celui-ci.
- le deuxième est le service Short Message Mobile Originated Point-to-Point (SMMO) qui gère la transmission du message entre le serveur de stockage et le portable destinataire.

Le service SMMT:
Le service fonctionne de manière suivante:

- L'émetteur remet le SMS à son centre de redirection des messages (MSC) de rattachement.
- Le MSC émet un message à son registre des visiteurs (VLR) pour lui demander le numéro de téléphone de l'émetteur et pour vérifier qu'aucune restriction n'est imposée à cet émetteur.
- Le VLR retourne alors une réponse.
- Si la réponse est positive, le MSC émet le message à l'interface réseau normalisé GSM (SMS-IWMSC).
- L'interface le retransmet à son tour au centre SMS (SMSC). Le SMSC stocke le message et les adresses dans sa mémoire.
- Le SMSC retourne une réponse (rapport de livraison) au SMS-IWMSC.
- Ce rapport est retourné par le SMS-IWMSC au MSC.
- Le MSC retourne à l'émetteur un message de rapport du statut du SMS.

Le service peut être schématisé de cette manière:

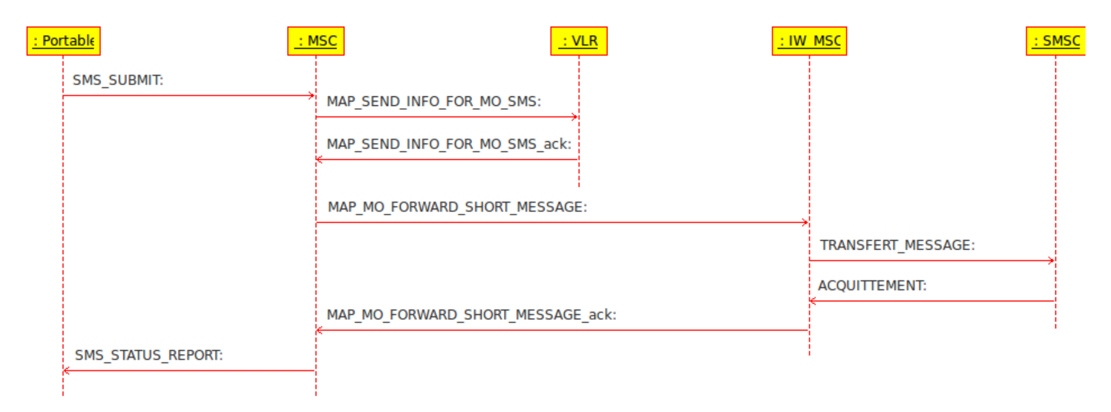


Figure 10: Ici, Alice fait confiance à Bob, car son certificat a été signé par une AC en qui elle a confiance. Elle fait également confiance à Charlie car elle peut trouver une AC qui fait à la fois confiance à son AC et à celle de Charlie

Le service SMMO:
Le service fonctionne de la manière suivante:

- L'interface GSM (SMS-GMSC) reçoit un message court du centre SMS (SMSC).
- L'interface demande des informations de routage du message au registre de localisation (HLR), qui lui permettent de relayer le message au centre de redirection (MSC) approprié (MSC auquel est rattachée la station mobile destinataire).

- Le HLR utilise ce numéro pour rechercher les informations de routage qu'il retourne au SMS-GMSC à travers la réponse. Cette réponse contient l'identifiant du destinataire (IMSI) et l'adresse du MSC de rattachement.
- Le SMS-GMSC délivre le message court au MSC.
- Le MSC émet une requête à son registre des visiteurs (VLR) en vue d'obtenir des informations relatives au destinataire. Le paramètre passé dans cette requête est l'IMSI du destinataire.
- A partir de l'IMSI fourni par le MSC, le VLR identifie la zone de localisation (LA) du mobile destinataire. Le VLR lance alors une procédure de paging (technique consistant à effectuer une recherche sur l'ensemble de la zone où est susceptible de se trouver le mobile demandé). Si le VLR ne connaît pas l'identité du destinataire, un message est alors émis afin de lancer la procédure de paging sur toutes les LA dépendant du MSC.
- Le MSC effectue la procédure de paging sur la zone de localisation du destinataire.
- La station mobile destinataire répond positivement.
- Le VLR retourne une réponse au MSC, autorisant ce dernier à relayer le message court à la station mobile destinataire.
- Le MSC achemine le message court au destinataire et reçoit un acquittement.
- Le MSC inclut ce rapport dans un réponse retourné au SMS-GMSC.
- Le SMS-GMSC passe le rapport au SMSC.

Le service peut être schématisé de cette manière:

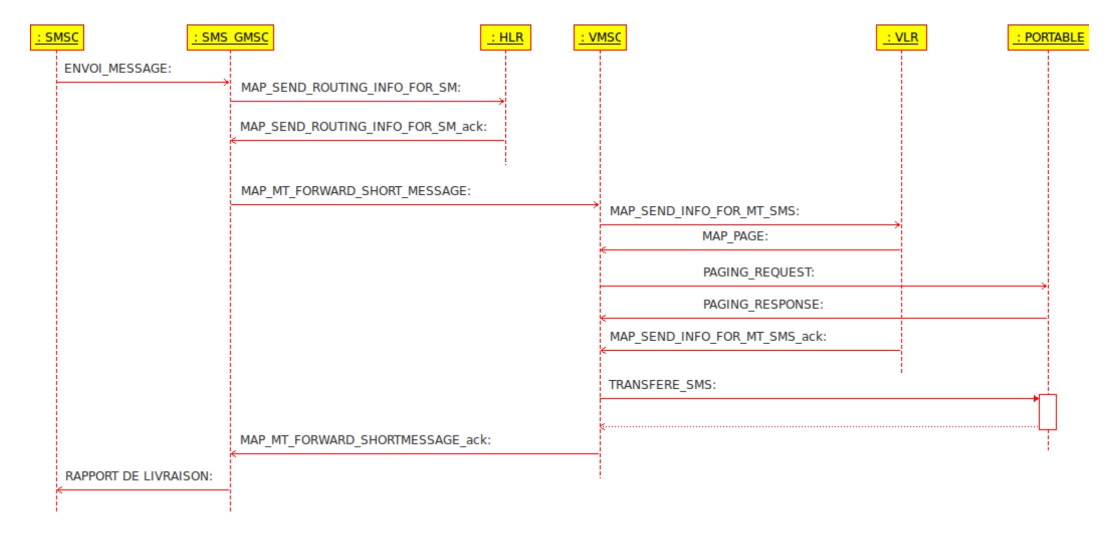


Figure 11: Ici, Alice fait confiance à Bob, car son certificat a été signé par une AC en qui elle a confiance. Elle fait également confiance à Charlie car elle peut trouver une AC qui fait à la fois confiance à son AC et à celle de Charlie

Envoi du SMS dans notre projet:
L'envoi des SMS depuis notre application est géré par la classe `SendSMSActivity`. Cette classe fera appel à la classe `SmsManager` du paquet android.telephony
Nous avons également créé notre propre `BroadcastReceiver` pour l'envoi des messages. Celui-ci nous permettra de vérifier la bonne émission des messages envoyés ou, dans le cas contraire, d'afficher un message d'erreur précis sur la cause du non envoi.

```
ArrayList<String> msgList=SmsManager.getDefault().divideMessage(msg);
ArrayList<PendingIntent> sentIntents= new ArrayList<PendingIntent>(msgList.size());
for (int i = 0; i < msgList.size(); i++){
        Intent sendInt = new Intent("ENVOYER");
        sendInt.putExtra("PART", "Partie "+i);
```

```
        PendingIntent sendResult = PendingIntent.getBroadcast(SendSMSActivity.this, 0, sendInt,
            PendingIntent.FLAG\_CANCEL\_CURRENT);
        sentIntents.add(sendResult);
}
SmsManager.getDefault().sendMultipartTextMessage(num, null, msgList, null, null);
```

Le message sera découpé en une liste de SMS pour que le destinataire le voit comme un seul message. Celui-ci sera ensuite envoyé grâce à la méthode `sendMultipartTextMessage` de la classe `SmsManager`. Les arguments de la fonction sont: le numéro du destinataire, le numéro de l'expéditeur (ou null pour utiliser le SMSC défini par défaut sur le téléphone), un `PendingIntent` permettant de vérifier l'émission du SMS et un pour vérifier sa transmission. Pour la création des `PendingIntent`, nous créons tout d'abord un `Intent` dont l'action sera ENVOYER. Nous rajoutons également dans l'`Intent` la partie du message concernée afin d'avoir des messages d'erreur plus précis. Enfin, nous créons un `PendingIntent` qui transmettra l'`Intent`. C'est cet `Intent` qui sera récupéré par le `BroadcastReceiver` que nous avons créé au début de la classe.

4 Schéma du Projet

4.1 Diagramme UML

Le projet se décompose en trois parties distinctes, qui sont la partie SMS, la partie cryptographique et la partie gestion des clefs. Nous en faisons la description ci-dessous ainsi que celle des messages à travers les différentes fonctionnalités.

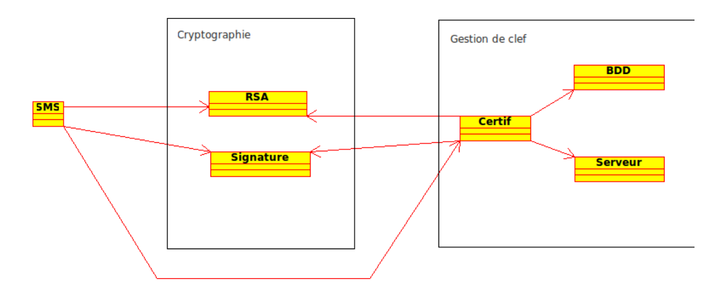


Figure 12: Schéma du projet

SMS Dans cette partie, nous gérons les services nécessaires pour les SMS, c'est à dire que c'est dans cette partie que nous allons récupérer le message à chiffrer, le numéro de téléphone du destinataire, que nous allons envoyer les SMS et que nous allons les lire, en les déchiffrant si besoin. Les fonctions associées à cette partie sont :

- La réception/lecture de SMS ;
- L'envoi de SMS ;
- La vérification du numéro de téléphone du propriétaire.

La partie SMS est en relation avec la classe RSA pour chiffrer ou déchiffrer un message, avec la classe Signer pour générer ou vérifier la signature d'un message et les classes Certificat, BaseDeDonnee et CommServeur pour la récupération des clefs.

Cryptographie La partie Cryptographique gère le chiffrement, le déchiffrement et la signature d'un message. Les fonctions associées a cette partie sont :

- La génération des clefs ;
- Le chiffrement de SMS ;
- Le déchiffrement de SMS ;
- La signature d'un SMS ;
- La vérification de signature.

Elle se décompose en deux classes, l'une pour le chiffrement et le déchiffrement nommée RSA, et la deuxième pour la gestion de la signature (création et vérification) appelée Signer.

Gestion des clefs La gestion de clef consiste à :

- Stocker notre clef publique sur un serveur dans le but de la diffuser ;
- Récupérer une clef publique pour envoyer un message ;
- Révoquer un certificat que l'on a mis en ligne.

Pour cela, nous avons les fonctions associées suivantes:

- Le stockage des certificats récupérés ;

- L'envoi d'un certificat à un serveur ;
- La récupération d'un certificat correspondant à un numéro de téléphone ;
- La révocation de certificat à partir du téléphone ou d'un ordinateur ;
- La transmission des clefs que l'on a récupérées.

Cette partie est composée de trois classes : la première est la classe Certificat qui va nous permettre de stocker toutes les données d'un certificat dans un seul objet, ensuite la classe BaseDeDonnee qui va gérer la base de donnée du téléphone où l'on stocke les certificats déjà récupérés et enfin la dernière classe qui s'occupe des échanges avec le serveur et qui est appelé CommServeur.

4.2 Détail des fonctionnalités

Pour notre projet, nous avons réfléchi à six fonctionnalités qui sont :

- L'envoi d'un SMS chiffré ;
- La lecture d'un SMS chiffré ;
- La création d'un compte ;
- La récupération et le stockage d'un certificat ;
- La mise à jour d'un certificat ;
- La révocation d'un certificat.

Envoi d'un SMS chiffré :

Pour envoyer un SMS chiffré, la classe SMS va demander à la classe BaseDeDonnee le certificat du destinataire. S'il n'y a pas de certificat dans la base de données, on le demande à la classe CommServeur et on vérifie la signature du certificat. Si celle-ci est correcte, on la stocke dans notre base de données. Sinon, on fait la mise à jour du certificat que l'on stocke sur la BDD. Puis elle retourne la clef à la classe SMS qui va envoyer une demande de chiffrement du message à la classe RSA.

Lecture d'un SMS chiffré :
Lors de la lecture d'un SMS chiffré et signé, la classe SMS va demander à la classe Signature de vérifier la signature du message. Pour cela nous devons récupérer le certificat dans la classe BaseDeDonnee. S'il n'y a pas de certificat, on le demande au Serveur. On vérifie alors la signature du certificat et si la elle est correcte, on stocke le certificat dans notre base.
Si nous avons un certificat dans note base, nous vérifions qu'il est encore à jour en envoyant sa signature au serveur. Celui-ci va alors comparer les 2 signatures et, si elles ne correspondent pas, renvoyer le nouveau certificat que nous allons alors stocker dans notre base.
Enfin, une fois ce certificat récupéré, nous en récupérons la clef publique avec la classe Certificat. Nous pouvons donc vérifier la signature de notre message et, si celle-ci est correcte, nous pourrons alors envoyer le message à déchiffrer à la classe RSA avec notre clef privée chargée depuis nos préférences.

Création d'un compte :
Pour créer un compte, l'utilisateur va permettre à la classe SMS de vérifier le numéro de téléphone pour lequel on va générer le certificat, puis l'utilisateur va demander à la classe Certificat de créer un certificat. Pour cela elle va demander à la classe RSA de générer les clefs puis va envoyer la demande au serveur de créer un compte et d'y associer le certificat précédemment généré. Pour cela, la classe CommServer va envoyer une requête contenant le pseudo, le mot de passe, le numéro de téléphone et le certificat au format XML. Si le pseudo du compte n'est pas déjà utilisé, nous ajoutons la clef privée dans nos préférences, sinon, on demande un autre pseudo à l'utilisateur.

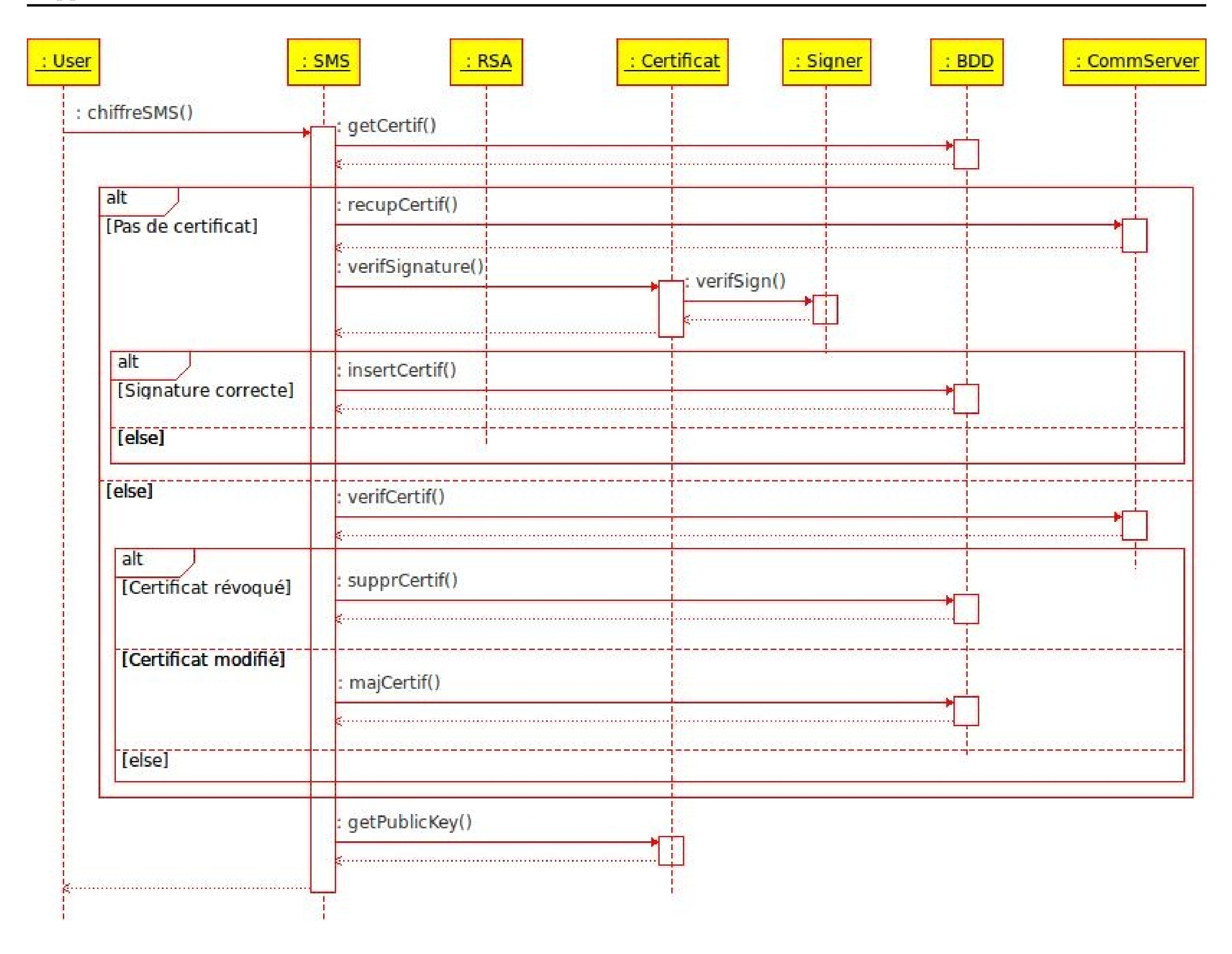


Figure 13: Échanges lors de l'envoi d'un SMS

Récupération et stockage d'un certificat :
Pour récupérer un certificat, l'utilisateur va envoyer une requête au serveur avec la classe CommServer contenant le numéro de téléphone du certificat à récupérer. Le serveur va alors vérifier qu'il possède dans sa base de donnée un certificat associé au numéro reçu. Si c'est le cas, il enverra alors ce certificat sous la forme d'une chaîne XML ainsi que la signature de ce certificat que la classe Signer pourra alors vérifier. Le format XML a été choisi pour sa simplicité d'utilisation, aussi bien pour analyser une chaîne XML que pour en construire une à partir d'un objet.
Le stockage des certificats s'effectue grâce aux classes BaseDeDonnee et BaseDeDonneesOpenHelper qui nous ont permis de créer notre propre table SQLite qui sera sauvegardée dans le téléphone. C'est dans cette table que nous entrerons les données contenues dans un certificat.

Mise à jour d'un certificat :
Pour mettre à jour un certificat, l'utilisateur va demander à la classe Certificat de créer un nouveau certificat. Pour cela elle va demander à la classe RSA de générer les clefs puis va envoyer la demande au Serveur de modifier le certificat du compte. Si la modification est validée (la connexion au compte est validée), nous modifions la clef privée dans nos préférences, sinon, on ne sauvegarde pas les changements.

Révocation d'un certificat :
Pour supprimer un certificat, l'utilisateur va envoyer une requête de suppression à notre serveur avec la classe CommServeur. La réponse à cette requête nous indiquera si l'opération de suppression a bien été effectuée : si c'est le cas, alors nous supprimons la clef privée de nos préférences, sinon, aucune modification n'est effectuée.

Figure 14: Échanges lors de la lecture d'un SMS

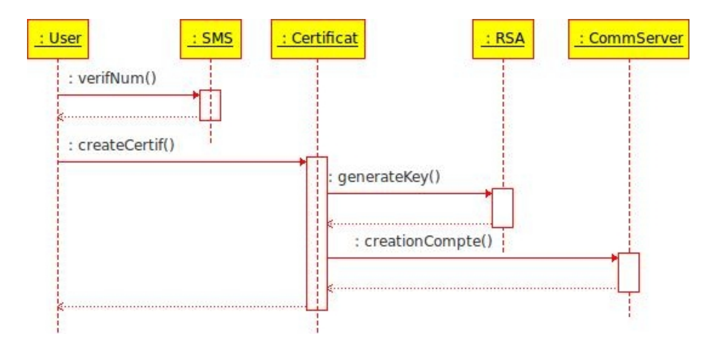


Figure 15: Échange lors de la création d'un compte

4.3 Explication de code

Classe MainActivity

Cette classe est celle qui va définir ce qu'on va voir lorsqu'on démarre l'application.
Lorsqu'un utilisateur ouvre l'application, deux cas se présentent : il peut s'agir d'une première utilisation ou non. Pour vérifier dans quel cas nous sommes, nous regardons si nous trouvons des préférences définies par l'application (ici, `modulo`, qui correspond au module de chiffrement des clefs) :

```
//charge le contenu du fichier de preference keys dans lequel sont stockees les clefs
SharedPreferences pref = this.getSharedPreferences("keys", Context.MODE_PRIVATE);
//pref.getString renvoie la valeur de modulo si la preference existe ou une chaine vide si ce n'est pas
    le cas
if(pref.getString("modulo", "").equals("")){...}
```

Si nous ne trouvons pas le modulo dans les préférences, nous appelons la classe `PremiereUtilisationActivity` qui se chargera entre autres de générer ces clefs.
La vue de cette classe dispose de deux boutons : *ecrire* et *lire*. Ces boutons appellent respectivement les classes `SendSMSActivity` (qui envoie les SMS) et `ReadSMSActivity` (qui lit les SMS).

Classe PremiereUtilisationActivity

Cette classe va nous servir à enregistrer un nouvel utilisateur. Afin de vérifier son identité, nous lui demandons d'entrer son numéro de téléphone, puis nous lui envoyons un SMS avec une valeur aléatoire et enfin, nous vérifions dans les SMS reçus s'il a bien cette valeur aléatoire provenant du numéro spécifié.
La méthode `boolean verifNum(String num)` va nous servir pour effectuer les actions concernant la vérification du numéro (génération de la valeur aléatoire, envoi de SMS et réception).

```
Random r=new Random();
String message=String.valueOf(r.nextInt(1000));

//on l'envoie
SmsManager.getDefault().sendTextMessage(num, null, message, null, null);

//on attend
try {
        Thread.sleep(5000);
} catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}

//on verifie qu'on l'a bien recu
Uri uriSMS = Uri.parse("content://sms/inbox");
String[] projection=new String[] {"body"};;
String selection="address=?";
String[] selectionArgs=new String[]{num};
Cursor curseur = getContentResolver().query(uriSMS, projection, selection, selectionArgs, "date DESC
    LIMIT 1");
if (curseur.moveToFirst()) {

        if(curseur.getString(curseur.getColumnIndex("body"))
          .equals(message)){
                return true;
        }
}
return false;
```

La génération du nombre aléatoire est effectuée grâce à la classe `Random` du paquet java.util. L'envoi du SMS se fait avec la classe `SmsManager`, comme dans la classe `SendSMSActivity` que nous avons étudié précédemment. Ici, nous n'utiliserons que le contenu du message et le numéro du destinataire.
Nous attendons ensuite cinq secondes pour recevoir le SMS. Cette valeur a été choisie afin d'avoir un temps d'attente assez long pour envoyer et recevoir un message tout en essayant de bloquer le moins possible notre programme. Pour la vérification de la réception, nous voulions au départ utiliser un `BroadcastReceiver` qui s'exécuterait lorsque le SMS serait reçu, mais il aurait fallu mettre un système de minuteur au bout duquel nous considérerions que le numéro donné était faux, ce qui était plus compliqué.
Nous récupérons ensuite avec une requête sur l'URI (pour Uniform Resource Identifier, soit identifiant uniforme de ressource) SMS le dernier message provenant du numéro donné. Si ce message vaut le nombre aléatoire généré, alors nous renvoyons vrai, sinon nous renvoyons faux. Si la méthode `verifyNum` nous renvoie vrai, nous générons alors la paire de clefs privée et publique que nous stockons dans les préférences.

```
SharedPreferences pref = this.getSharedPreferences("keys", Context.MODE_PRIVATE);
final KeyPair kp=RSA.generateKeyPair();
RSAPublicKey pub=(RSAPublicKey)kp.getPublic();
RSAPrivateKey priv=(RSAPrivateKey)kp.getPrivate();
SharedPreferences.Editor editor = pref.edit();
editor.putString("numero", num);
editor.putString("modulo", pub.getModulus().toString());
editor.putString("exposant_public", pub.getPublicExponent().toString());
editor.putString("exposant_prive", priv.getPrivateExponent().toString());
editor.commit();
```

Les préférences sont gérées par la classe `SharedPreferences`. Nous récupérons celles contenues dans le fichier "keys" afin de pouvoir en rajouter.
Les clefs sont générées à l'aide de notre classe RSA. Afin de pouvoir charger les clefs pour les utiliser pour déchiffrer ou chiffrer un message, nous avons du les stocker sous la forme d'un modulo et d'un exposant public ou privé selon la clef. Pour stocker une nouvelle préférence, nous devons récupérer un éditeur à l'aide de la méthode `edit()`. C'est dans cet éditeur que nous allons ajouter les préférences une par une, sous la forme d'un couple (clef, valeur). Une fois les préférences ajoutées, il ne reste plus qu'à les valider.
Après la génération des clefs, nous retournons à la classe `MainActivity`.

Classe ListContact

Cette classe va nous permettre de générer une liste des contacts enregistrés avec leur numéro de téléphone.
Pour récupérer des ressources du téléphone, nous devons déjà récupérer leur identifiant (URI).

```
Uri contactUri = ContactsContract.Contacts.CONTENT_URI;
Uri phoneUri = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
```

Le premier URI nous servira à récupérer les noms des contacts et le second leur numéro.

```
String[] projection = new String[] { ContactsContract.Contacts._ID,
ContactsContract.Contacts.DISPLAY_NAME,
ContactsContract.Contacts.HAS_PHONE_NUMBER};
String selection = ContactsContract.Contacts.HAS_PHONE_NUMBER + "=?";
String[] selectionArgs = new String[]{"1"};
Cursor contacts = ctx.getContentResolver().query(contactUri, projection, selection, selectionArgs, null)
    ;
```

Nous effectuons ensuite une requête sur l'URI des contacts. Le tableau `projection` correspond aux colonnes que nous voulons récupérer, ici : l'identifiant du contact, le nom du contact et s'il a au moins un numéro de téléphone ou non. La chaîne `selection` nous permet d'apporter une condition sur la colonne indiquant si le contact à un numéro. Cette condition est précisée juste en dessous dans le tableau `selectionArgs` qui contient la liste des arguments de la sélection (un argument est identifié par le caractère '?' dans la sélection). Le contact doit donc avoir au moins un numéro de téléphone. La ligne suivante permet de récupérer le résultat de la requête ainsi créée. Le dernier argument de cette requête est pour l'ordre des lignes du résultat (l'équivalent d'ORDER BY en SQL), ce que nous n'utiliserons pas pour cette requête.

```
String[] phone_projection = new String[] { ContactsContract.CommonDataKinds.Phone.NUMBER };
String phone_selection = ContactsContract.CommonDataKinds.Phone.CONTACT_ID+ "=?";
String[] phone_selectionArgs;
do {
        String contactId = contacts.getString(contacts.getColumnIndex(ContactsContract.Contacts._ID));
    phone_selectionArgs = new String[]{contactId};
    Cursor phones = ctx.getContentResolver().query(phoneUri, phone_projection, phone_selection,
        phone_selectionArgs, ContactsContract.CommonDataKinds.Phone.IS_SUPER_PRIMARY + " DESC");
    if (phones.moveToFirst()) {
            do {
                    try {
                            number = phones.getString(phones.getColumnIndex(ContactsContract.
                                CommonDataKinds.Phone.NUMBER));
                            name = contacts.getString(contacts.getColumnIndex(ContactsContract.
                                Contacts.DISPLAY_NAME));
                            list.add(name + " , " + number);
                    }
                    catch (Exception e) {
                            Log.i(this.getClass().getName(), e.getMessage());
                    }
            } while (phones.moveToNext());
            phones.close();
        }
} while (contacts.moveToNext());
```

Le problème que nous rencontrons ici est qu'un contact peut avoir plusieurs numéros. Nous devons donc, pour chaque contact, récupérer la liste de ses numéros. Pour cela, nous allons effectuer une requête sur l'URI correspondant aux numéros de téléphone. Cette requête aura les arguments suivants : pour la projection, nous ne récupérons que le numéro de téléphone; pour la sélection, nous nous servirons de l'identifiant récupéré dans la requête précédente pour n'avoir que les numéros du contact courant ; pour l'ordre, nous allons faire en sorte de faire apparaître le numéro principal du contact en premier : IS_SUPER_PRIMARY est une constante qui vaut 1 si le numéro est le principal, 0 sinon.
Une fois que nous avons un curseur contenant le résultat de la requête, il ne nous reste plus qu'à le stocker avec le nom du contact dans la liste finale.

Classe SendSMSActivity

La vue de cette classe comprend trois boutons et deux zones de texte : une pour le destinataire et une pour le message à envoyer. Afin de faciliter l'envoi d'un message à un de nos contacts, nous avons utilisé une zone de texte qui s'auto-complète, nous devons donc charger la liste des contacts, ce qui se fait dans la classe `ListContact`.
Si l'utilisateur appuie sur le bouton `encode` ("Chiffrer"), nous vérifions alors que nous disposons de la clef publique de l'expéditeur dans notre base de données en faisant appel à la classe `BaseDeDonnee`. Si c'est le cas, nous vérifions qu'il s'agit bien de la bonne clef grâce à la fonction `signatureOK` de la classe `CommServer`, sinon nous récupérons directement le certificat de l'expéditeur grâce à la fonction `recupCertif` de la même classe. Une fois la clef récupérée, nous chiffrons le message grâce à la fonction `RSAEncode` de la classe `RSA`.
Si l'utilisateur appuie sur le bouton *signe* ("Signer") et si le message est chiffré, nous chargeons notre clef privée et signons le message grâce à la méthode `signeMess` de la classe `Signer`. Dans le cas contraire, nous affichons un message indiquant à l'utilisateur qu'il doit d'abord chiffrer le message, ceci afin de permettre de vérifier l'intégrité et la provenance d'un message avant de tenter de le déchiffrer.
Enfin, si l'utilisateur appuie sur le bouton `envoie` ("Envoyer"), nous récupérons les données entrées dans les champs `dest` et `text` correspondant respectivement au destinataire et au message et nous envoyons le message en utilisant le code présenté lors de l'analyse de l'envoi d'un message.

Classe ReadSMSActivity

Cette classe va nous servir à afficher la liste des contacts qui nous ont envoyé des SMS. En cliquant sur un de ces contacts, on pourra alors afficher les SMS dont il est l'expéditeur.
Nous utiliserons la méthode `public ArrayList<HashMap<String, String>>SMStoList()` afin de récupérer la liste des contacts sous la forme d'une liste de couple (noms, numéro).

```
Uri uriSMS = Uri.parse("content://sms/inbox");
Uri uriPhone = ContactsContract.CommonDataKinds.Phone.CONTENT_URI;
Uri uriContact = ContactsContract.Contacts.CONTENT_URI;
```

Nous aurons besoin de 3 URIs : la première pour obtenir les SMS reçus, la seconde pour obtenir l'id d'un contact à partir de son numéro de téléphone et enfin la dernière pour trouver le nom d'un contact à partir de son id.

```
String[] projection = new String[] {"DISTINCT address"};
Cursor curseur = getContentResolver().query(uriSMS, projection, null, null, null);
```

La première requête que nous effectuons va nous permettre de récupérer tous les numéros des personnes qui nous ont envoyé des SMS. Nous récupérons le champ `address` qui correspond au numéro de téléphone de l'expéditeur.

```
HashMap<String, String> map = new HashMap<String, String>();
String address=curseur.getString(curseur.getColumnIndex("address"));
//l'adresse recuperee contient l'indicatif pays qui n'est pas forcement dans Phone.NUMBER
String selectionPhone = ContactsContract.CommonDataKinds.Phone.NUMBER + " LIKE ?";
String[] selectionArgsPhone = new String[]{"\%"+address.substring(3)};
String[] projectionPhone=new String[]{ContactsContract.CommonDataKinds.Phone.CONTACT_ID};
Cursor phoneCursor = getContentResolver().query(uriPhone, projectionPhone, selectionPhone,
        selectionArgsPhone, null);
```

Le numéro récupéré contient toujours l'indicatif pays (+33 pour un numéro français par exemple), or, le numéro stocké dans la table `Phone` est celui rentré par l'utilisateur, donc il ne contient pas forcément cet indicatif. C'est pourquoi nous avons du utiliser une sélection avec le mot clef "LIKE" et couper le numéro de téléphone pour supprimer l'indicatif.
Nous effectuons une requête sur la table `Phone` afin de récupérer l'identifiant du contact pour retrouver son nom.

```
String name="";
String id=phoneCursor.getString(phoneCursor.getColumnIndex(
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID));
String selectionContact=ContactsContract.Contacts._ID + "=?";
String[] selectionArgsContact=new String[]{id};
String[] projectionContact=new String[]{ContactsContract.Contacts.DISPLAY_NAME};
Cursor contactCursor=getContentResolver().query(uriContact, projectionContact,
        selectionContact, selectionArgsContact, null);
if(contactCursor.moveToFirst()){
        name= contactCursor.getString(contactCursor.getColumnIndex(ContactsContract.Contacts.
            DISPLAY_NAME));
        contactCursor.close();
}
map.put("Numero", address);
map.put("Nom", name);
```

Nous effectuons enfin la dernière requête sur la table Contact afin de récupérer le nom de l'expéditeur. Nous allons donc chercher le nom (DISPLAY_NAME) du contact dont l'identifiant (_ID) correspond à l'identifiant récupéré précédemment (CONTACT_ID). Si nous obtenons un résultat, nous le stockons dans une collection (map) que nous ajouterons à la liste de contacts à renvoyer.

```
ArrayList<HashMap<String, String>> mylist = SMStoList();
SimpleAdapter adapter = new SimpleAdapter(this, mylist,
R.layout.listeboitereceptionelement_layout, new String[] {"Nom", "Numero"},
        new int[] {R.id.nom,R.id.numero});
list.setAdapter(adapter);
```

Une fois cette liste récupérée, nous créons un adaptateur pour pouvoir remplir la `listView` que l'utilisateur voit sur l'écran. Pour cela, nous construisons un nouveau `SimpleAdapter` avec pour argument l'activité dans laquelle le `SimpleAdapter` va être, la liste contenant les données sous forme de map, le layout qui va définir comment se fera l'affichage, une liste des noms de colonnes qui seront associés à chaque élément de la collection et les vues qui correspondront à chacune de ces colonnes (il s'agit ici de `textView` définies dans le layout `listeboitereceptionelement_layout`).

```
HashMap<String, String> map = (HashMap<String, String>) list.getItemAtPosition(position);
Intent intent = new Intent(ReadSMSActivity.this, ReadSMSContactActivity.class);
intent.putExtra("address", map.get("Numero"));
startActivity(intent);
```

Enfin, lorsque l'on clique sur un des contacts de la liste, nous récupérons la map associée (contenant le nom et le numéro du contact) et nous créons un nouvel `Intent` qui va indiquer que l'on veut maintenant exécuter la classe `ReadSMSContactActivity`. Avant de démarrer l'`Intent`, nous passons également le numéro du contact qui nous servira pour récupérer les messages envoyé par ce numéro.

Classe ReadSMSContactActivity

Cette classe va servir à afficher tous les SMS d'un même contact.
La méthode `ArrayList<HashMap<String,String>>SMStoList(String address)` va nous servir à récupérer ces SMS.

```
Uri uriSMS = Uri.parse("content://sms/inbox");
String selection = "address=?";
String[] selectionArgs=new String[]{address};
String[] projection = new String[] {"date", "body"};
Cursor curseur = getContentResolver().query(uriSMS, projection, selection, selectionArgs, "date DESC
    LIMIT 10");
```

Nous préparons donc notre requête pour récupérer les SMS : nous voulons que l'expéditeur soit le numéro passé en argument de la méthode, et nous voulons récupérer la date et le contenu du message. Nous renseignons également un ordre : du plus récent au plus ancien, et une limite que nous avons fixée à 10 pour éviter que le chargement soit trop long.

```
SharedPreferences pref = this.getSharedPreferences("keys", Context.MODE_PRIVATE);
BigInteger modulo=new BigInteger(pref.getString("modulo", null));
BigInteger priv=new BigInteger(pref.getString("exposant_prive", null));
RSAPrivateKeySpec privateKeySpec=new RSAPrivateKeySpec(modulo,priv);
KeyFactory keyFactory = KeyFactory.getInstance("RSA");
privKey = (PrivateKey) keyFactory.generatePrivate(privateKeySpec);
```

Nous chargeons ensuite notre clef privée pour pouvoir déchiffrer les messages reçus si besoin. Cette clef est stockée dans les préférences sous la forme d'un modulo et d'un exposant, ceci afin de pouvoir créer directement notre clef à partir du constructeur de la classe `RSAPrivateKeySpec`. Nous pourrons alors, si le message est signé, vérifier sa signature à l'aide de la fonction `verifieSignature` de la classe `Signer` ; et, si nous trouvons un message commençant par l'octet 127 (indiquant qu'il est chiffré), appeler la fonction `RSADecode` de notre classe RSA avec ce message et avec la clef récupérée.

```
//on recupere l'address de l'expediteur
Intent thisIntent = getIntent();
address = thisIntent.getExtras().getString("address");
ArrayList<HashMap<String, String>> mylist = SMStoList(address);
SimpleAdapter adapter = new SimpleAdapter(this, mylist,
        R.layout.listesmselement_layout, new String[] {"Date", "Message"}, new int[] {R.id.date,R.id.
            body});
list.setAdapter(adapter);
```

Dans la méthode principale, nous devons tout d'abord récupérer le numéro passé par la classe `ReadSMSActivity` avec la clef `address`. Ensuite, une fois la liste des SMS récupérée grâce à la méthode SMStoList, nous créons un adaptateur comme nous l'avions fait dans la classe `ReadSMSActivity` avec les colonnes `date` et `message`.

Classe Certificat

Cette classe va définir un objet `Certificat`. Cet objet a pour champs : `numero` pour le numéro de téléphone, `modulo` et `exposant` pour la clef publique, `algorithme` pour indiquer quel algorithme utiliser (seul RSA est supporté pour le moment) et `sign` pour la signature du certificat.
La méthode `public Certificat fromXML(String xml)` va nous permettre de convertir une chaîne XML en un objet `Certificat`. En effet, lorsque nous demandons un certificat au serveur, il nous le renverra sous la forme d'une chaîne XML afin de faciliter la vérification de la signature.

```
Certificat c=new Certificat();
SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser sp;
try {
        sp = spf.newSAXParser();
        XMLReader xr = sp.getXMLReader();
        CertificatHandler dataHandler = new CertificatHandler();
        xr.setContentHandler(dataHandler);
        xr.parse(new InputSource(new StringReader(xml)));
        c = dataHandler.getCertif();
}
```

Pour analyser une chaîne XML, nous utiliserons la librairie SAX et une classe que nous avons créée : `CertificatHandler` qui définit le comportement de l'analyseur lorsqu'il rencontre un tag. Elle nous servira de `contentHandler` (soit de gestionnaire de contenu). Nous devons donc construire un `XMLReader` afin de pouvoir lui passer la chaîne XML pour qu'il l'analyse. Pour cela, nous devons construire un `SAXParser` qui est lui même généré par un `SAXParserFactory`.

Classe CertificatHandler

Cette classe étend `DefaultHandler` qui est la classe par défaut pour gérer les événements SAX. Elle va nous permettre de définir les actions de l'analyseur lorsqu'il commencera l'analyse du document, lorsqu'il rencontrera un tag particulier...
La première des méthodes que nous modifions est la méthode `public void startDocument()` qui est appelée lorsque l'analyseur commence à lire un nouveau document (dans notre cas une chaîne de caractère). Dans cette méthode, nous allons juste initialiser `inCertif` et `inClef` qui sont des booléens que nous utiliserons pour vérifier que le certificat est bien formaté.
La méthode `startElement(String uri, String localName, String name, Attributes attributes)` est appelée lorsque l'analyseur commence à lire un nouvel élément (il rencontre une nouvelle balise). Le nom de cette balise est contenu dans `localName`.

```
if (localName.equalsIgnoreCase("certificat")){
        this.c = new Certificat();
        inCertif = true;
}
```

Ici, si nous sommes dans un tag `certificat`, nous initialisons un nouvel objet `Certificat` qui contiendra les données que nous lirons et nous initialisons `inCertif` à vrai pour indiquer que nous sommes dans un élément `certificat`.
La méthode `characters(char[] ch, int start, int length)` va récupérer les caractères contenus entre 2 balises. Ces caractères sont stockés dans le tableau `ch` de l'indice `start` à `length`.

```
String lecture = new String(ch,start,length);
if(buffer != null)
        buffer+=lecture;
```

Dans notre objet `CertificatHandler`, nous avons une String `buffer` qui contiendra les caractères lus. Cette chaîne est remise à null dès que nous finissons de lire un élément.
Enfin, la méthode `endElement(String uri, String localName, String name)` est appelée lorsque l'analyseur lit une balise fermante. C'est dans cette méthode que nous remplissons les champs du certificat selon la balise rencontrée.

```
if (localName.equalsIgnoreCase("numero")){
        if(inCertif){
                this.c.setNumero(buffer.toString());
                buffer = null;
        }
}
```

Par exemple, si nous sommes dans une balise fermante "numero", qui contient le numéro de téléphone, nous savons que celui-ci est contenu dans `buffer`, nous pouvons donc mettre le champ `numero` de `Certificat` à la valeur trouvée dans `buffer`.

Classe BaseDeDonneesOpenHelper

La classe `BaseDeDonneesOpenHelper` étend la classe `SQLiteOpenHelper` du paquet android.database.sqlite. Cette classe va se charger de la création de notre base de donnée si elle n'existe pas déjà, de sa mise à jour et de son ouverture pour accéder aux données.
Nous devons ré-implémenter deux méthodes ainsi que le constructeur de cette classe. Pour ce dernier, nous avons simplement utilisé le constructeur de `SQLiteOpenHelper`.

```
private final String TABLE_CERTIF = "table_certificat";
private final String COL_NUMERO = "Numero";
private final String COL_SIGN = "Signature";
private final String COL_MOD = "Modulo";
private final String COL_EXP = "Exposant";
private final String COL_ALGO = "Algorithme";

private final String CREATE_BDD = "CREATE TABLE " + TABLE_CERTIF +
        " (" + COL_NUMERO + " TEXT PRIMARY KEY, "
        + COL_ALGO + " TEXT NOT NULL, "
        + COL_MOD + " TEXT NOT NULL, "
        + COL_EXP + " TEXT NOT NULL, "
        + COL_SIGN + " TEXT);";
```

Nous avons également défini des String pour la table, le nom des colonnes et la requête de création de la table.
La méthode `void onCreate(SQLiteDatabase db)` est la méthode appelée lorsque la base de donnée est créée. C'est donc ici que nous allons créer notre table que nous allons appeler `table_certificat`.

```
db.execSQL(CREATE_BDD);
```

Nous devons seulement appeler la méthode `execSQL` qui va exécuter la requête décrite précédemment et ainsi créer la table qui contiendra les certificats.
La deuxième méthode implémentée est `void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` qui va être appelée si, lors de la mise à jour de notre application, nous avons besoin de changer la structure de la table. Il suffira alors de changer le numéro de version passé en argument dans le constructeur.

```
db.execSQL("DROP TABLE IF EXISTS " + TABLE_CERTIF + ";");
onCreate(db);
```

Pour le moment nous supprimons simplement la table pour la recréer ensuite.

Classe BaseDeDonnee

C'est dans cette classe que nous allons accéder aux données de notre table `table_certificat`.

```
private final int VERSION_BDD = 1;
private final String NOM_BDD = "ter.db";
private final String TABLE_CERTIF = "table_certificat";
private final String COL_NUMERO = "Numero";
private final String COL_SIGN = "Signature";
private final String COL_MOD = "Modulo";
private final String COL_EXP = "Exposant";
private final String COL_ALGO = "Algorithme";

private SQLiteDatabase bdd;
private BaseDeDonneesOpenHelper bddoh;

public BaseDeDonnee(Context context){
        //On cree la BDD et sa table
        bddoh = new BaseDeDonneesOpenHelper(context, NOM_BDD, null, VERSION_BDD);
        bdd = bddoh.getWritableDatabase();
}
```

Comme pour la classe `BaseDeDonneesOpenHelper`, nous avons définis des String contenant le nom de nos colonnes et celui de la table. Nous avons également défini un champ `SQLiteDatabase bdd` qui va nous servir pour récupérer ou pour modifier les données de notre base et un `BaseDeDonneesOpenHelper bddoh` pour créer, si besoin, la base de donnée et pour la récupérer.
Le constructeur de notre classe va créer un nouvel objet `BaseDeDonneesOpenHelper` qui va nous servir à initialiser notre objet `bdd`. La base de données ainsi que la table seront ainsi créées si besoin. Nous demandons les accès écriture à la base car nous vérifions à chaque envoi si le certificat n'a pas été modifié, et s'il l'a été, nous devons le mettre à jour. Il y a donc toujours une possibilité pour que nous ayons à écrire dans la base de données.
La méthode `long insertCertificat(Certificat c)` va nous servir à insérer un nouveau certificat dans la table.

```
ContentValues values = new ContentValues();
values.put(COL_NUMERO, c.getNumero());
values.put(COL_SIGN, c.getSign());
values.put(COL_ALGO, c.getAlgorithme());
values.put(COL_EXP, c.getExposant().toString());
values.put(COL_MOD, c.getModulo().toString());
return bdd.insert(TABLE_CERTIF, null, values);
```

Nous utilisons la classe `ContentValues` pour stocker les couples (nom de colonne, valeur) que nous allons insérer dans la table. Cette insertion se fait grâce à la fonction `insert` qui prend en argument la table, une chaîne qui indique la colonne à mettre à NULL si l'on veut insérer une ligne vide (que l'on n'utilisera pas ici), et un objet `ContentValues` contenant les valeurs à insérer.
La méthode `int updateCertif(String numero, Certificat c)` va nous servir à mettre à jour un certificat, lorsque celui-ci a été révoqué par exemple.

```
String[] whereArgs=new String[]{numero};
return bdd.update(TABLE_CERTIF, values, COL_NUMERO + "␣=␣?", whereArgs);
```

La méthode pour mettre à jour une table est `update`, qui prend en argument la table, les nouvelles valeurs associées aux colonnes correspondantes (un objet `ContentValues` utilisé de la même façon que dans la méthode précédente), une String pour indiquer le contenu de la clause WHERE dans la requête SQL et un tableau de String pour les valeurs dans cette clause (remplacées par '?' dans la chaîne sélection).
La méthode `int removeCertif(String numero)` va nous permettre de supprimer un certificat de la table.

```
String[] whereArgs=new String[]{numero};
return bdd.delete(TABLE_CERTIF, COL_NUMERO + "␣=␣?", whereArgs);
```

Cette suppression se fait grâce à la méthode `delete` qui prend en argument la table, le contenu de la clause WHERE et un tableau contenant les valeurs de cette clause.
La méthode `Certificat cursorToCertif(Cursor curs)` va nous être utile pour que, lorsque l'on recherche un certificat dans la table, le résultat nous soit rendu sous la forme d'un `Certificat` et non pas sous celle d'un `Cursor`.

```
if (!curs.moveToFirst())
        return null;

//On cree un certificat
Certificat cert = new Certificat();
cert.setNumero(curs.getString(curs.getColumnIndex(COL_NUMERO)));
cert.setAlgorithme(curs.getString(curs.getColumnIndex(COL_ALGO)));
cert.setModulo(new BigInteger(curs.getString(curs.getColumnIndex(COL_MOD))));
cert.setExposant(new BigInteger(curs.getString(curs.getColumnIndex(COL_EXP))));
cert.setSign(curs.getString(curs.getColumnIndex(COL_SIGN)));

curs.close();
```

Le seul appel à cette requête est fait depuis la fonction `getCertificat` qui doit renvoyer le certificat correspondant à un numéro. Le résultat de la requête correspondante ne contiendra donc qu'un seul certificat. C'est pourquoi nous ne parcourons pas le curseur et ne regardons que la première ligne. Nous remplissons ensuite notre objet Certificat grâce aux manipulateurs (les méthodes commençant par set) que nous avons définis dans la classe Certificat.
Enfin, la dernière méthode, `Certificat getCertif(String numero)`, va nous permettre de récupérer le certificat d'un contact.

```
String[] columns= new String[] {COL_NUMERO, COL_EXP, COL_MOD, COL_ALGO, COL_SIGN};
String selection= COL_NUMERO + " =?";
String[] selectionArgs= new String[]{numero};
Cursor curs = bdd.query(TABLE_CERTIF, columns, selection, selectionArgs, null, null, null);
```

Nous construisons donc notre requête : la sélection se fera sur la colonne `numero`, dont la valeur devra être égale au `numero` passé en argument. Les 3 derniers paramètres de la méthode `query` correspondent respectivement aux GROUP BY, HAVING et ORDER BY que nous n'utiliserons pas ici.

Classe CommServer

C'est dans cette classe que se font toutes les communications avec le serveur. Le format utilisé pour les réponses du serveur est le format JSON qui a l'avantage d'être simple à mettre en place : il existe deux classes en Java qui nous permettront de traiter directement les données reçues, les classes JSONArray et JSONObject du paquet org.json, et il existe une fonction en PHP (qui est le langage utilisé pour notre serveur) pour transformer un tableau en un objet de type JSON: json_encode.
Cette classe dispose de plusieurs méthodes qui correspondent à la création des différentes requêtes que nous pouvons envoyer ainsi qu'à l'analyse de leur réponse. Elle dispose également d'une méthode qui va effectuer l'envoi d'une requête. Cette méthode a pour signature `public static JSONObject execRequete(String url, ArrayList<NameValuePair>nvp)`.

```
DefaultHttpClient httpclient = new DefaultHttpClient();
HttpPost hp = new HttpPost(url);
hp.setEntity(new UrlEncodedFormEntity(nvp));

HttpResponse response = httpclient.execute(hp);
HttpEntity entity = response.getEntity();
is = entity.getContent();

BufferedReader reader = new BufferedReader(new InputStreamReader(is,"ISO-8859-1"));
StringBuilder sb = new StringBuilder();
String line = null;
while ((line = reader.readLine()) != null){
        sb.append(line + "\n");
}
is.close();
result = sb.toString();
```

L'argument url de la méthode correspond donc à l'url de la page de notre serveur traitant la requête : par exemple, creation_compte.php pour la création d'un nouveau compte. Le deuxième argument est un tableau de couple (clef, valeur) correspondant aux paramètres de la requête, ceux qui seront récupérés sur le serveur par un $_POST[clef] (puisque nous utilisons la méthode POST).
Pour l'envoi de la requête, nous nous servons de classes disponible dans le paquet org.apache.http. Les quatres classes majeures sont `DefaultHttpClient`, `HttpPost`, pour l'envoi de la requête ; `HttpResponse` et `HttpEntity` pour récupérer la réponse du serveur. Cette réponse sera ensuite convertie en chaîne de caractère puis en un objet JSON. Les champs de cet objet seront ensuite récupéré à l'aide de la méthode `getString(String clef)` de l'objet `JSONObject`.

5 Programme

5.1 Manuel d'utilisation

Accueil

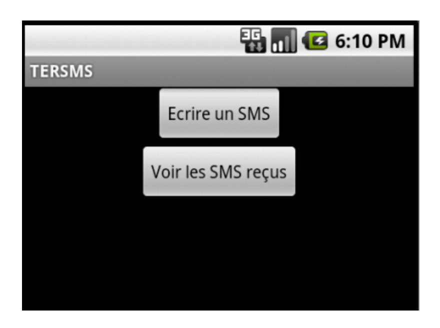


Figure 16: Page d'accueil

La page d'accueil est la page que l'utilisateur découvre lorsqu'il ouvre l'application. A partir de cette page, l'utilisateur peut choisir d'envoyer un message en appuyant sur "Ecrire un SMS" ou de lire ses sms en appuyant sur "Voir les SMS reçus". Cependant, lorsque l'utilisateur ouvre l'application pour la première fois, c'est une autre page qui va s'afficher, lui permettant de créer un nouveau certificat.

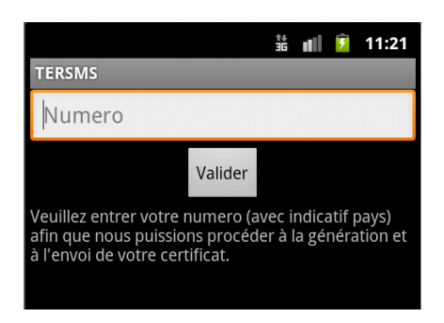


Figure 17: Générer son certificat

L'utilisateur doit donc rentrer son numéro (avec l'indicatif pays) et appuyer sur Valider. Une fois ceci fait, il devra alors attendre quelques secondes afin de savoir si son numéro a été vérifié ou non. Au cours de ce temps d'attente, il recevra un sms contenant seulement un chiffre qu'il ne faudra pas supprimer: il s'agit de la vérification.
Une fois la vérification faite, l'application retournera automatiquement à la page d'accueil. Si la vérification n'a pas pu être effectuée, l'utilisateur verra alors une pop-up s'afficher lui indiquant que son numéro n'a pas été accepté.

Ecriture d'un SMS

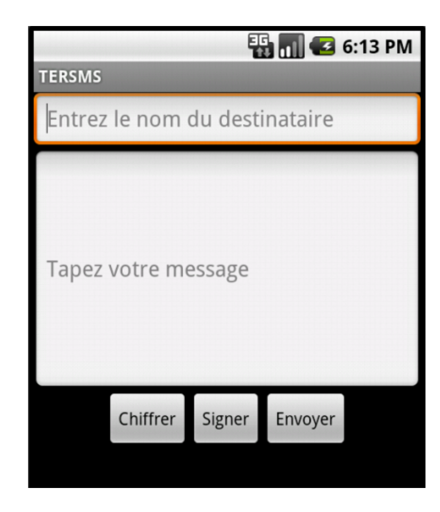


Figure 18: Ecrire un SMS

L'écriture d'un message se fait en 2 temps. Premièrement, l'utilisateur va écrire son message ainsi que choisir le destinataire.

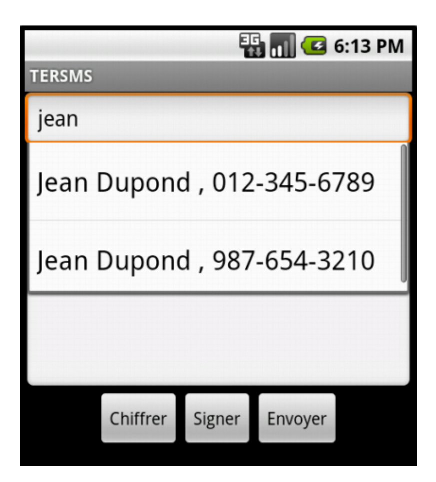


Figure 19: Choisir le destinataire

Pour choisir son destinataire, l'utilisateur peut taper les lettres du prénom ou du nom du contact pour qu'une liste de propositions apparaisse. Il peut également choisir de rentrer directement un numéro de téléphone si son destinataire n'est pas enregistré dans son téléphone.
Ensuite, il va pouvoir choisir de chiffrer le message, le signer (seulement si le message est chiffré) ou alors l'envoyer. Lorsque l'utilisateur chiffre ou signe son message, le contenu du message se met à jour dans l'affichage. Il est donc impossible de le modifier par la suite.

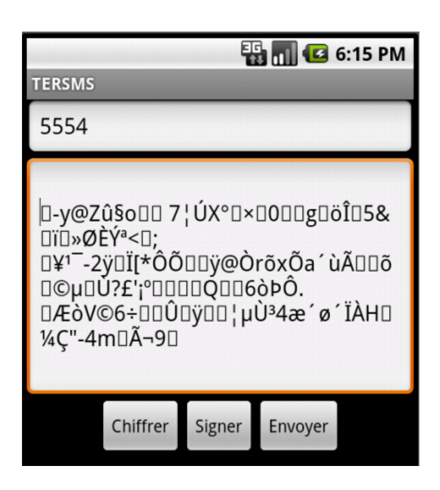


Figure 20: Chiffrer un SMS

Lecture d'un SMS

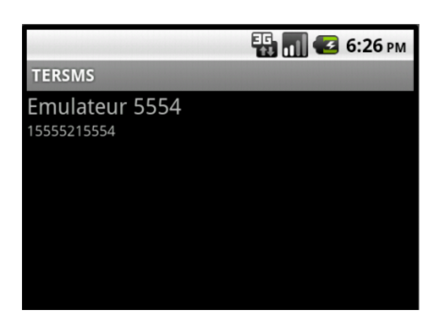


Figure 21: Liste des contacts

Lorsque l'utilisateur choisit "Voir les SMS reçus", il arrive alors sur une page listant tous les contacts qui lui ont envoyé un message.
En cliquant sur un de ces contacts, l'utilisateur accède alors à la liste des derniers sms reçus de ce contact. Il peut ainsi voir la date du sms et son contenu (déchiffré si besoin). Cette liste est pour le moment limité aux 10 derniers sms afin de limiter le temps de chargement.
L'utilisateur peut également cliquer sur le bouton "Répondre" qui l'enverra alors sur la page "Ecrire un SMS" avec le champ destinataire initialisé au numéro du contact.



Figure 22: Liste des SMS

5.2 Parties implémenté

Actuellement le programme permet d'envoyer un SMS chiffré et signé, de récupérer un certificat sur le serveur au format XML ou d'en vérifier la validité (Mise à Jour ou Révocation) de plus les certificats sont stockés dans une base de donnée interne au portable permettant une utilisation prochaine plus rapide. Enfin une fois le SMS reçu, on peut vérifier la signature de celui-ci et si elle est correct, nous le déchiffrons pour le lire.
Mais la connexion https avec le serveur ne fonctionne pas, le téléphone portable nous signale qu'il n'y a pas de certificat et les algorithmes de signature du certificat ne sont pas les mêmes que ce présent sur Android, de ce fait les signatures testées ne sont pas reconnu, or nous ne savons pas ce qui différencie les deux algorithmes.
Le travail restant est l'implémentation d'un nouveau mode d'opération (différent de l'ECB), de créer un serveur plus sécurisé...

5.3 Améliorations possibles

Les améliorations que nous avions envisagées étaient de pouvoir transférer un certificat par SMS sans avoir à passer par le serveur, de pouvoir transférer les SMS sécurisés vers un autre portables en changeant le chiffré et enfin de pouvoir transférer sa clef privé via le Bluetooth par exemple.

5.4 Le problème rencontré

L'organisation L'organisation du temps a été une difficulté importante du projet, au fil des semaine, nous nous sommes rendu-compte que nous perdons du temps parce que nous nous dispersions en effectuant plusieurs tache en même temps.

5.5 Apport du projet

Ce projet nous a permis d'apprendre la programmation Android, d'illustrer et d'approfondir nos connaissances sur les infrastructures à clef publique et la sécurité WEB, et enfin l'importance des schémas UML en début de projet.

6 Conclusion

Avec ce projet, nous pouvons désormais utilisé le service de SMS pour envoyer des informations confidentielles ou alors développer les services de paiement par SMS et pourquoi pas imaginé d'utiliser notre programme pour permettre d'utiliser notre téléphone comme une carte bancaire via le service SMS (envoi d'un demande de virement...)

Liste des figures

Liste des tableaux

Liste des algorithmes

Références bibliographiques

[AAR] University College London Gower Street London WC1E 6BT United Kingdom Alfarez Abdul-Rahman, Department of ComputerScience. The pgp trust model. http://www.wim.uni-koeln.de/uploads/media/The_PGP_Trust_Model.pdf.

[All07a] Open Handset Alliance. Industry leaders announce open platform for mobile devices, 2007. http://www.openhandsetalliance.com/press_110507.html.

[All07b] Open Handset Alliance. Open handset alliance releases android sdk, 2007. http://www.openhandsetalliance.com/press_111207.html.

[Ben] Mustapha Benjada. Pki (public key infrastructure). http://www.securiteinfo.com/cryptographie/pki.shtml.

[Buc06] Johannes Buchmann. *Introduction à la cryptographie*. Dunod, 2006.

[CAc] CAcert.org. Cacert wiki. http://wiki.cacert.org/.

[Dev12] Android Developers. Platform versions, 2012. http://developer.android.com/resources/dashboard/platform-versions.html.

[eCW11] Laurent Bloch et Christophe Wolfhugel. *Sécurité Informatique: Principes et méthode*. Eyrolles, 2011.

[eFeT09] Etude en FORmation en Télécommunication. Short message service: Principe et architecture, 2009. http://www.efort.com/r_tutoriels/SMS_EFORT.pdf.

[Fou04] The Eclipse Foundation. History of eclipse, 2004. http://www.eclipse.org/org/#history.

[Pie] Laurent Piechocki. Uml en francais. http://uml.free.fr/.